LSF Programmer's Guide

Version 3.2
Fourth Edition, August 1998

Platform Computing Corporation

LSF Programmer's Guide

Printed in Canada

Revision Information for LSF Programmer’s Guide

Edition	Description
First	This document describes the Application Programming Interfaces of LSF version 2.2
Second	Revised and redesigned to describe LSF 3.0
Third	Revised and redesigned to describe LSF 3.1
Fourth	Revised to describe the new features of LSF 3.2

Contents

Preface

Audience

This guide provides tutorial and reference information for programmers who want to create programs that use the features of the Load Sharing Facility (LSF) software.

You should be familiar with the concepts described in the *LSF User's Guide* as well as with C programming in UNIX and/or Windows NT environments. If you are going to write programs using the calendars and events of the LSF JobScheduler, you should also be familiar with the *LSF JobScheduler User's Guide*.

LSF Suite 3.2

LSF is a suite of workload management products including the following:

LSF Batch is a batch job processing system for distributed and heterogeneous environments, which ensures optimal resource sharing.

LSF JobScheduler is a distributed production job scheduler that integrates heterogeneous servers into a virtual mainframe or virtual supercomputer

LSF MultiCluster supports resource sharing among multiple clusters of computers using LSF products, while maintaining resource ownership and cluster autonomy.

LSF Analyzer is a graphical tool for comprehensive workload data analysis. It processes cluster-wide job logs from LSF Batch and LSF JobScheduler to produce statistical reports on the usage of system resources by users on different hosts through various queues.

LSF Parallel is a software product that manages parallel job execution in a production networked environment.

LSF Make is a distributed and parallel Make based on GNU Make that simultaneously dispatches tasks to multiple hosts.

LSF Base is the software upon which all the other LSF products are based. It includes the network servers (LIM and RES), the LSF API, and load sharing tools.

There are two editions of the LSF Suite:

LSF Enterprise Edition

Platform's LSF Enterprise Edition provides a reliable, scalable means for organizations to schedule, analyze, and monitor their distributed workloads across heterogeneous UNIX and Windows NT computing environments. LSF Enterprise Edition includes all the features in LSF Standard Edition (LSF Base and LSF Batch), plus the benefits of LSF Analyzer and LSF MultiCluster.

LSF Standard Edition

The foundation for all LSF products, Platform's Standard Edition consists of two products, LSF Base and LSF Batch. LSF Standard Edition offers users robust load sharing and sophisticated batch scheduling across distributed UNIX and Windows NT computing environments.

Related Documents

The following guides are available from Platform Computing Corporation:

LSF Installation Guide
LSF Batch Administrator's Guide
LSF Batch Administrator's Quick Reference
LSF Batch User's Guide
LSF Batch User's Quick Reference
LSF JobScheduler Administrator's Guide
LSF JobScheduler User's Guide

LSF Analyzer User's Guide
LSF Parallel User's Guide
LSF Programmer's Guide

Online Documentation

- Man pages (accessed with the `man` command) for all commands
- Online help available through the Help menu for the `xlsbatch`, `xbmod`, `xbsub`, `xbalarms`, `xbcal` and `xlsadmin` applications.

Technical Assistance

If you need any technical assistance with LSF, please contact your reseller or Platform Computing's Technical Support Department at the following address:

LSF Technical Support
Platform Computing Corporation
3760 14th Avenue
Markham, Ontario
Canada L3R 3T7

Tel: +1 905 948 8448
Toll-free: 1-87PLATFORM (1-877-528-3676)
Fax: +1 905 948 9975
Electronic mail: *support@platform.com*

Please include the full name of your company.

You may find the answers you need from Platform Computing Corporation's home page on the World Wide Web. Point your browser to *www.platform.com*.

If you have any comments about this document, please send them to the attention of LSF Documentation at the address above, or send email to *doc@platform.com*.

1 Introduction

This chapter gives an overview of the LSF system architecture and the load sharing services provided by the LSF API, introducing their components. It also demonstrates how to write, compile, and link a simple load sharing application using LSF.

LSF Product Suite and Architecture

LSF is a layer of software services on top of UNIX and Windows NT operating systems. LSF creates a single system image on a network of heterogeneous computers such that the whole network of computing resources can be utilized effectively and managed easily. Throughout this guide, LSF refers to the LSF suite, which contains the following products:

LSF Base

LSF Base provides the basic load-sharing services across a heterogeneous network of computers. It is the base software upon which all other LSF functional products are built. It provides services such as resource information, host selection, placement advice, transparent remote execution and remote file operation, etc.

LSF Base includes Load Information Manager (LIM), Remote Execution Server (RES), the LSF Base API, `lstools` that allow the use of LSF Base to run simple load-sharing applications, `lstcsh`, and `lsmake`.

LSF Batch

LSF Batch is a distributed batch queuing system built on top of the LSF Base. The services provided by LSF Batch are extensions to the LSF Base services. LSF Batch makes a computer network a network batch computer. It has all the features of a mainframe batch job processing system while doing load balancing and policy-driven resource allocation control.

LSF Batch relies on services provided by the LSF Base. It makes use of the resource and load information from the LIM to do load balancing. LSF Batch also uses the cluster configuration information from LIM and follows the master election service provided by LIM. LSF Batch uses RES for interactive batch job execution and uses the remote file operation service provided by RES for file transfer. LSF Batch includes a Master Batch Daemon (`mbatchd`) running on the master host and a slave Batch Daemon (`sbatchd`) running on each batch server host.

LSF JobScheduler

LSF JobScheduler is a network production job scheduling system that automates the mission-critical activities of a MIS organization. It provides reliable job scheduling on a heterogeneous network of computers with centralized control. LSF JobScheduler reacts to calendars and events to schedule jobs at the correct time on the correct machines.

Like LSF Batch, LSF JobScheduler is built on top of the LSF Base. It relies on LSF Base in resource matching, job placement, cluster configuration, and distributed file operation. LSF JobScheduler supports calendars, file events, and user defined events in scheduling production jobs.

LSF MultiCluster

LSF MultiCluster extends the capabilities of the LSF system by sharing the resources of an organization across multiple cooperating clusters of computers. Load sharing happens not only within the clusters but also among them. Resource ownership and autonomy is enforced, non-shared user accounts and file systems are supported, and communication limitations among the clusters are also considered in job scheduling.

LSF consists of a number of servers running as root on each participating host in an LSF cluster and a comprehensive set of utilities built on top of the LSF Application Programming Interface (API). The LSF API consists of two libraries:

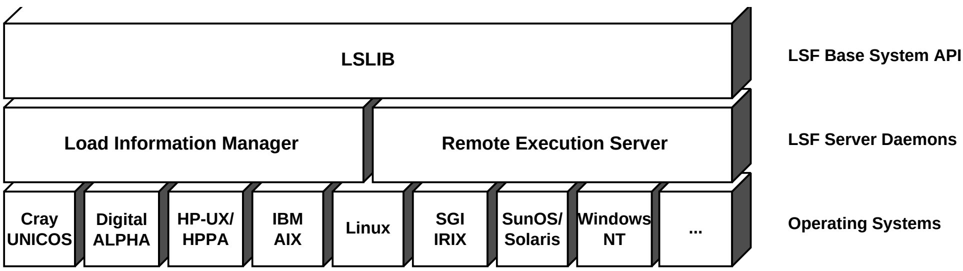
- Basic LSF services are accessible to applications through LSLIB, the LSF Base library.

- Job scheduling and processing services are accessible through LSF Batch library, LSBLIB. This library allows applications to get services from LSF Batch and LSF JobScheduler.

LSF Base System

Figure 1 shows the components of the LSF Base and their relationship.

Figure 1. LSF Base Architecture



LSF Base consists of two servers, the Load Information Manager (LIM*)* and the Remote Execution Server (RES), and the Load Sharing Library (LSLIB). LSF Base provides the basic load sharing services across a heterogeneous network of computers.

An LSF server host is a host that runs load-shared jobs. The LIM and RES run on every LSF server host. They interface directly with the underlying operating systems and provide users with a uniform, host independent environment.

One of the LIMs acts as the master. The master LIM is chosen among all the LIMs running in the cluster based on the configuration file settings. If the master LIM becomes unavailable, the LIM on the next configured host will automatically take over.

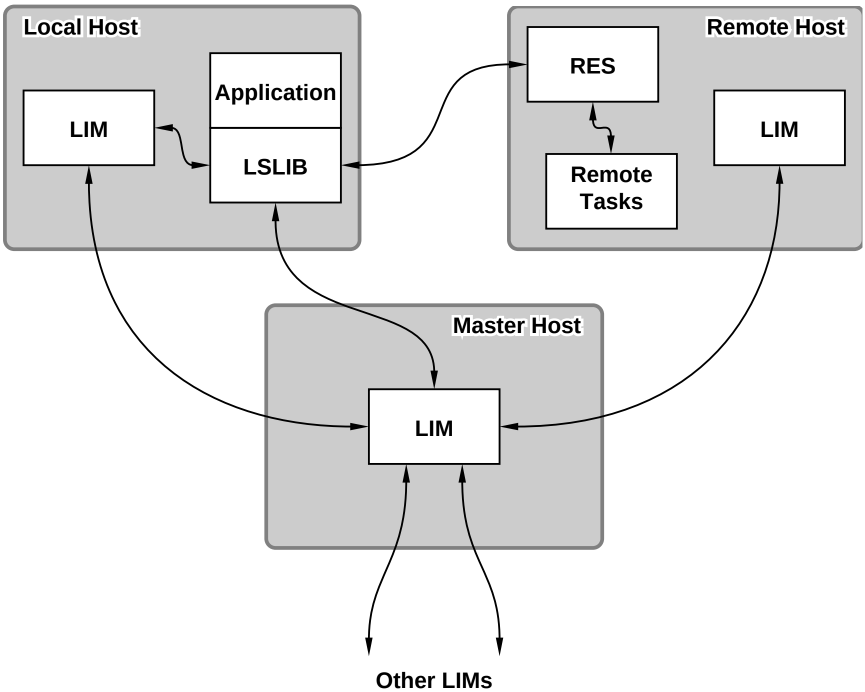
The LIM on each host monitors its host's load and reports load information to the master LIM. The master LIM collects information from all hosts and provides that information to the applications.

The RES on each server host accepts remote execution requests and provides fast, transparent, and secure remote execution of tasks.

Application and LSF Base Interactions

The diagram below shows the typical interactions between an LSF application and the LSF Base.

Figure 2. LIM, RES, LSLIB and Applications



In order to find out the information about the LSF clusters, an application calls the information service functions in the LSLIB which then contact the LIM to get the information. If the information requested is only available from the master LIM, then LSLIB will automatically send the request to the master host.

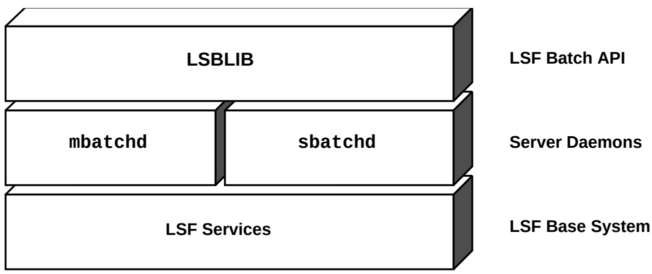
To run a task remotely, or to perform a file operation remotely, an application calls the remote execution or remote file operation service functions in the LSLIB, which then contact the RES to get the services.

The LIM on individual machines communicate periodically to update the information they provide to the applications.

LSF Batch System

LSF Batch is a layered distributed load sharing batch system built on top of the LSF Base. The services provided by LSF Batch are extensions to the LSF Base services. Application programmers can access the batch services through the LSF Batch Library, LSBLIB.

Figure 3. Structure of LSF Batch System

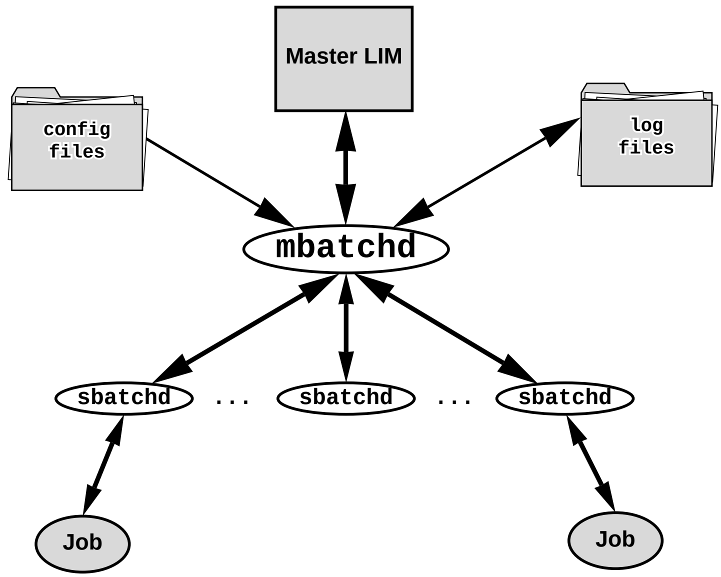


LSF Batch accepts user jobs and holds them in queues until suitable hosts are available. LSF Batch runs user jobs on LSF Batch server hosts, those hosts that a site deems suitable for running batch jobs.

LSF Batch services are provided by one `mbatchd` (master batch daemon) running in each LSF cluster, and one `sbatchd` (slave batch daemon) running on each batch server host.

LSF Batch operation relies on the services provided by the LSF Base. LSF Batch contacts the master LIM to get load and resource information about every batch server host.

Figure 4. The Operation of LSF Batch System



`mbatchd` always runs on the host where master LIM runs. The `sbatchd` on the master host automatically starts the `mbatchd`. If the master LIM moves to a different host, the current `mbatchd` will automatically resign and a new `mbatchd` will be automatically started on the new master host.

User jobs are held in batch queues by `mbatchd`, which checks the load information on all candidate hosts periodically. When a host with the necessary resources becomes available, `mbatchd` sends a job to the `sbatchd` on that host for execution. When more than one host is available, the best host is chosen.

Once a job is sent to an `sbatchd`, that `sbatchd` controls the execution of the job and reports job status to `mbatchd`.

The log files store important system and job information so that a newly started `mbatchd` can restore the status of the previous `mbatchd` easily. The log files also provide historic information about jobs, queues, hosts, and LSF Batch servers.

LSF JobScheduler System

LSF JobScheduler shares the same architecture and job processing mechanism. In addition to services provided by LSF Batch, LSF JobScheduler also provides calendar and event processing services. Both LSF Batch and LSF JobScheduler provides API to applications via LSBLIB.

Note

In the reminder of this Guide, all descriptions about LSF Batch also apply to LSF JobScheduler unless explicitly stated otherwise.

LSF API Services

LSF services are natural extensions to operating system services. LSF services glue heterogeneous operating systems into a single, integrated computing system.

LSF APIs provide easy access to the services of LSF servers. The API routines hide the details of interactions between the application and LSF servers in a way that is platform independent.

LSF APIs have been used to build numerous load sharing applications and utilities. Some examples of applications built on top of the LSF APIs are `lsmake`, `lstcsh`, `lsrun`, LSF Batch user interface, and `xlsmon`.

LSF Base API Services

The LSF Base API (LSLIB) allows application programmers to get services provided by LIM and RES. The services include:

Configuration Information Service

This set of function calls provide information about the LSF cluster configuration, such as hosts belonging to the cluster, total amount of installed resources on each host (for example, number of CPUs, amount of physical memory, and swap space), special resources associated with individual hosts, and types and models of individual hosts.

Such information is static and is collected by LIMs on individual hosts. By calling these routines, an application gets a global view of the distributed system. This information can be used for various purposes. For example, the LSF command `lshosts` displays such information on the screen. LSF Batch also uses such information to know how many CPUs are on each host.

Flexible options are available for an application to select the information that is of interest to it.

Dynamic Load Information Service

This set of function calls provide comprehensive dynamic load information collected from individual hosts periodically. The load information is provided in the form of load indices detailing the load on various resources of each host, such as CPU, memory, I/O, disk space, and interactive activities. Since a site-installed External LIM (ELIM) can be optionally plugged into the LIM to collect additional information that is not already collected by the LIM, this set of services can be used to collect virtually any type of dynamic information about individual hosts.

Example applications that use such information include `lsload`, `lsmon`, and `xlsmon`. This information is also valuable to an application in making intelligent job scheduling decisions. For example, LSF Batch uses such information to decide whether or not a job should be sent to a host for execution.

These service routines provide powerful mechanism for selecting the information that is of interest to the application.

Placement Advice Service

LSF Base API provides functions to select the best host among all the hosts. The selected host can then be used to run a job or to login to. LSF provides flexible syntax for an application to specify the resource requirements or criteria for host selection and sorting.

Many LSF utilities use these functions for placement decisions, such as `lsrun`, `lsmake`, and `lslogin`. It is also possible for an application to get the detailed load information about the candidate hosts together with a preference order of the hosts.

A parallel application can ask for multiple hosts in one LSLIB call for the placement of a multi-component job.

The performance differences between different models of machines as well as the number of CPUs on each host are taken into consideration when placement advice is made, with the goal of selecting qualified host(s) that will provide the best performance.

Task List Manipulation Service

Task lists are used to store default resource requirements for users. LSF provides functions to manipulate the task lists and retrieve resource requirements for a task. This is important for applications that need to automatically pick up the resource requirements from user's task list. The LSF commands `lsrtasks` uses these functions to manipulate user's task list. LSF utilities such as `lstcsh`, `lsrun,` and `bsub` automatically pick up the resource requirements of the submitted command line by calling these LSLIB functions.

Master Selection Service

If your application needs some kind of fault tolerance, you can make use of the master selection service provided by the LIM. For example, you can run one copy of your application on every host and only allow the copy on the master host to be the primary copy and others to be backup copies. LSLIB provides a function that tells you the name of the current master host.

LSF Batch uses this service to achieve improved availability. As long as one host in the LSF cluster is up, LSF Batch service will continue.

Remote Execution Service

The remote execution service provides a transparent and efficient mechanism for running sequential as well as parallel jobs on remote hosts. The services are provided by the RES on the remote host in cooperation with the Network I/O Server (NIOS) on the local host. The NIOS is a per application stub process that handles the details of the terminal I/O and signals on the local side. NIOS is always automatically started by the LSLIB as needed.

RES runs as root and runs tasks on behalf of all users in the LSF cluster. Proper authentication is handled by RES before running a user task.

LSF utilities such as `lsrun`, `lsgrun`, `ch`, `lsmake`, and `lstcsh` use the remote execution service.

Remote File Operation Service

The remote file operation service allows load sharing applications to operate on files stored on remote machines. Such services extend the UNIX and Windows NT file operation services so that files that are not shared among hosts can also be accessed by distributed applications transparently.

LSLIB provides routines that are extensions to the UNIX and Windows NT file operations such as `open`(2), `close`(2), `read`(2), `write`(2), `fseek`(3), `stat`(2), etc.

The LSF utility `lsrcp` is implemented with the remote file operation service functions.

Administration Service

These set of function calls allow application programmers to write tools for administrating the LSF servers. The operations include reconfiguring the LSF clusters, shutting down a particular LSF server on some host, restarting an LSF server on some host, turning logging on or off, locking/unlocking a LIM on a host, etc.

The `lsadmin` and `xlsadmin` utilities use the administration services.

LSF Batch API Services

The LSF Batch API, LSBLIB, gives application programmers access to the job queueing processing services provided by the LSF Batch servers. All LSF Batch user interface utilities are built on top of LSBLIB. The services that are available through LSBLIB include:

LSF Batch System Information Service

This set of function calls allow applications to get information about LSF Batch system configuration and status. These include host, queue, and user configurations and status.

The batch configuration information determines the resource sharing policies that dictate the behavior of the LSF Batch scheduling.

The system status information reflects the current status of hosts, queues, and users of the LSF Batch system.

Example utilities that use the LSF Batch configuration information services are `bhosts`, `bqueues`, `busers`, `bparams`, and `xlsbatch`.

Job Manipulation Service

The job manipulation service allows LSF Batch application programmers to write utilities that operate on user jobs. The operations include job submission, signaling, status checking, checkpointing, migration, queue switching, and parameter modification.

Log File Processing Service

LSBLIB provides convenient routines for handling log files used by LSF Batch. These routines return the records logged in the `lsb.events` and `lsb.acct` files. The records are stored in well-defined data structures.

The LSF Batch commands `bhist` and `bacct` are implemented with these routines.

LSF Batch Administration Service

This set of function calls are useful for writing LSF Batch administration tools. The LSF Batch command `badmin` is implemented with these library calls.

Calendar Manipulation Service

These library calls are used only if you are using the Production Job Scheduler of LSF (LSF JobScheduler). These function calls allow programmers to write utilities that create, check, or change LSF Batch calendars. All the calendar-related user interface commands of the LSF JobScheduler make use of the calendar manipulation functions of the LSF Batch API.

Getting Started with LSF Programming

LSF programming is like any other system programming. You are assumed to have UNIX and/or Windows NT operating system and C programming knowledge to understand the concepts involved.

`lsf.conf` File

This guide frequently refers to the file, `lsf.conf`, for the definition of some parameters. `lsf.conf` is a generic reference file containing definitions of directories and parameters. It is by default installed in `/etc`. If it is not installed in `/etc`, all users of LSF must set the environment variable `LSF_ENVDIR` to point to the directory in which `lsf.conf` is installed. Refer to 'LSF Base Configuration Reference' in the LSF Administrator's Guide for more details about the `lsf.conf` file.

LSF Header Files

All LSF header files are installed in the directory `LSF_INCLUDEDIR/lsf`, where `LSF_INCLUDEDIR` is defined in the file `lsf.conf`. You should include `LSF_INCLUDEDIR` in the include file search path, such as that specified by the '`-Idir`' option of some compilers or pre-processors.

There is one header file for LSLIB, the LSF Base API, and one header file for LSBLIB, the LSF Batch API.

`lsf.h`

An LSF application must include `<lsf/lsf.h>` before any of the LSF Base API services are called. `lsf.h` contains definitions of constants, data structures, error codes, LSLIB function prototypes, macros, etc., that are used by all LSF applications.

`lsbatch.h`

An LSF Batch application must include `<lsf/lsbatch.h>` before any of the LSF Batch API services are called. `lsbatch.h` contains definitions of constants, data structures, error codes, LSBLIB function prototypes, macros, etc., that are used by all LSF Batch applications.

Note

There is no need to explicitly include `<lsf/lsf.h>` *in an LSF Batch application because* `lsbatch.h` *already includes* `<lsf/lsf.h>`.

Linking Applications with LSF APIs

LSF API functions are contained in two libraries: `liblsf.a` (LSLIB) and `libbat.a` (LSBLIB) for all UNIX platforms. For Windows NT, the file names of these libraries are: `liblsf.lib` (LSLIB) and `libbat.lib` (LSBLIB), respectively. These files are installed in `LSF_LIBDIR`, where `LSF_LIBDIR` is defined in the file `lsf.conf`.

Note

LSBLIB is not independent by itself. It must always be linked together with LSLIB. This is because LSBLIB services are built on top of LSLIB services.

LSF uses BSD sockets for communications across the network. On systems that have both System V and BSD programming interfaces, LSLIB and LSBLIB typically use the BSD programming interface. On System V-based versions of UNIX, for example Solaris, it is normally necessary to link applications using LSLIB or LSBLIB with the BSD compatibility library. On Windows NT, a number of libraries are needed to be linked together with LSF API. Details of these additional linkage specifications are shown in the table below.

Table 1. Additional Linkage Specifications by Platform

Platform	Additional Linkage Specifications
ULTRIX 4	(none)
Digital UNIX	`-lmach -lmld`
HP-UX	`-lBSD`
AIX	`-lbsd`

Table 1. Additional Linkage Specifications by Platform

Platform	Additional Linkage Specifications
IRIX 5	`-lsun -lc_s`
IRIX 6	(none)
SunOS 4	(none)
Solaris 2	`-lnsl -lelf -lsocket -lrpcsvc -lgen`
NEC	`-lnsl -lelf -lsocket -lrpcsvc -lgen`
Sony NEWSs	`-lc -lnsl -lelf -lsocket -lrpcsvc -lgen -lucb`
ConvexOS	(none)
Cray Unicos	(none)
Linux	(none)
Windows NT	`-MT -DWIN32 libcmt.lib oldnames.lib kernel32.lib advapi32.lib user32.lib wsock32.lib mpr.lib netapi32.lib`

Note

On Windows NT, you need to add paths specified by `LSF_LIBDIR` *and* `LSF_INCLUDEDIR` *in* `lsf.conf` *to the environment variables* `LIB` *and* `INCLUDE`*, respectively.*

The `$LSF_MISC/examples` directory contains a makefile for making all the example programs in that directory. You can modify this file for use with your own programs.

All LSLIB function call names start with '`ls_`', whereas all LSBLIB function call names start with '`lsb_`'.

Error Handling

LSF API uses error numbers to indicate an error. There are two global variables that are accessible from the application. These variables are used in exactly the same way UNIX system call error number variable `errno` is used. The error number should only be tested when an LSLIB or LSBLIB call fails.

lserrno

An LSF program should test whether an LSLIB call is successful or not by checking the return value of the call instead of `lserrno`.

When any LSLIB function call fails, it sets the global variable `lserrno` to indicate the cause of the error. The programmer can either call `ls_perror()` to print the error message explicitly to the `stderr`, or call `ls_sysmsg()` to get the error message string corresponding to the current value of `lserrno`.

Possible values of `lserrno` are defined in `lsf.h`.

lsberrno

This variable is very similar to `lserrno` except that it is set by LSBLIB whenever an LSBLIB call fails. Programmers can either call `lsb_perror()` to find out why an LSBLIB call failed or use `lsb_sysmsg()` to get the error message corresponding to the current value of `lsberrno`.

Possible values of `lsberrno` are defined in `lsbatch.h`.

Note

> `lserrno` *and* `lsberrno` *should be checked only if an LSLIB or LSBLIB call fails respectively.*

Example Applications

Example Application using LSLIB

```
#include <stdio.h>
#include <lsf/lsf.h>

void main()
{
    char *clustername;

    clustername = ls_getclustername();
    if (clustername == NULL) {
```

```
        ls_perror("ls_getclustername");
        exit(-1);
    }

    printf("My cluster name is: <%s>\n", clustername);
    exit(0);
}
```

This simple example gets the name of the LSF cluster and prints it on the screen. The LSLIB function call `ls_getclustername()` returns the name of the local cluster. If this call fails, it returns a `NULL` pointer. `ls_perror()` prints the error message corresponding to the most recently failed LSLIB function call.

The above program would produce output similar to the following:

```
% a.out
My cluster name is: <test_cluster>
```

Example Application using LSBLIB

```
#include <stdio.h>
#include<lsf/lsbatch.h>

main()
{
    struct parameterInfo *parameters;

    if (lsb_init(NULL) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    parameters = lsb_parameterinfo(NULL, NULL, NULL);
    if (parameters == NULL) {
        lsb_perror("lsb_parameterinfo");
        exit(-1);
    }

    /* Got parameters from mbatchd successfully. Now print out the fields */
    printf("Job acceptance interval: every %d dispatch turns\n",
                    parameters->jobAcceptInterval);
    /* Code that prints other parameters goes here */
```

```
        /* ... */
    exit(0);
}
```

This example gets the LSF Batch parameters and prints them on the screen. The function `lsb_init()` must be called before any other LSBLIB function is called.

The data structure `parameterInfo` is defined in `lsbatch.h`.

Authentication

LSF programming is distributed programming. Since LSF services are provided network-wide, it is important for LSF to deliver the service without compromising the system security.

LSF supports several user authentication protocols. Support for these protocols are described in the section 'Remote Execution Control' of the LSF Administrator's Guide. Your LSF administrator can configure the LSF cluster to use any of the protocols supported.

Note that only those LSF API function calls that operate on user jobs, user data, or LSF servers require authentication. Function calls that return information about the system do not need to be authenticated. LSF API calls that must be authenticated are identified in 'List of LSF API Functions' on page 99.

The most commonly used authentication protocol, the privileged port protocol, requires that load sharing applications be installed as setuid programs. This means that your application has to be owned by root with the suid bit set.

If you need to frequently change and relink your applications with LSF API, you can consider using the `ident` protocol which does not require applications to be setuid programs.

2 Programming with LSLIB

This chapter provides simple examples that demonstrate the use of LSLIB functions in an application. The function prototypes as well as data structures that are used by the functions are described. Many of the examples resemble the implementation of the existing LSF utilities.

Getting Configuration Information

One of the services that LSF provides to applications is cluster configuration information service. This section describes how to get such services with a C program using LSLIB.

Getting General Cluster Configuration Information

In the previous chapter, a very simple application was introduced that prints the name of the LSF cluster. This section extends that example to print out more information about the LSF cluster, namely, the current master host name and the defined resource names in the cluster. It uses the following additional LSLIB function calls:

```
struct lsInfo *ls_info()
char *ls_getmastername()
```

The function `ls_info()` returns a pointer to the following data structure (as defined in `<lsf/lsf.h>`):

```
struct lsinfo {
    int     nRes;                              Number of resources in the system
    struct resItem *resTable;                  A resItem for each resource in the system
    int     nTypes;                            Number of host types
    char    hostTypes[MAXTYPES][MAXLSFNAMELEN];    Host types
    int     nModels;                           Number of host models
```

```
    char    hostModels[MAXMODELS][AXLSFNAMELEN]; Host models
    float   cpuFactor[MAXMODELS];                CPU factors of each host model
    int     numIndx;                             Total number of load indices in resItem
    int     numUsrIndix;                         Number of user-defined load indices
};
```

The function `ls_getmastername()` returns a string containing the name of the current master host.

Both of these functions return `NULL` on failure and set `lserrno` to indicate the error.

The `resItem` structure describes the valid resources defined in the LSF cluster:

```
struct resItem {
    name[MAXLSFNAMELEN];            The name of the resource
    char des[MAXRESDESLEN];         The description of the resorce
    enum valueType valueType;       BOOLEAN, NUMERIC, STRING
    enum orderType orderType;       INCR, DECR, NA
    int  flags;                     RESF_BUILTIN | RESF_DYNAMIC | RESF_GLOBAL
    int  interval;                  The update interval for a load index, in seconds
};
```

The constants `MAXTYPES`, `MAXMODELS`, and `MAXLSFNAMELEN` are defined in <`lsf/lsf.h`>. `MAXLSFNAMELEN` is the maximum length of a name in the LSF system.

A host type in LSF refers to a class of hosts that are considered to be compatible from an application point of view. This is entirely configurable, although normally hosts with the same architecture (binary compatible hosts) should be configured to have the same host type.

A host model in LSF refers to a class of hosts with the same CPU performance. The CPU factor of a host model should be configured to reflect the CPU speed of the model relative to other host models in the LSF cluster.

Below is an example program that displays the general LSF cluster information using the above LSLIB function calls.

```
#include <stdio.h>
#include <lsf/lsf.h>

main()
{
```

```
    struct lsInfo *lsInfo;
    char *cluster, *master;
    int i;

    cluster = ls_getclustername();
    if (cluster == NULL) {
        ls_perror("ls_getclustername");
        exit(-1);
    }
    printf("My cluster name is <%s>\n", cluster);
    master = ls_getmastername();
    if (master == NULL) {
        ls_perror("ls_getmastername");
        exit(-1);
    }
    printf("Master host is <%s>\n", master);
    lsInfo = ls_info();
    if (lsInfo == NULL) {
        ls_perror("ls_info");
        exit(-1);
    }
    printf("\n%-15.15s %s\n", "RESOURCE_NAME", "DESCRIPTION");
    for (i=0; i<lsInfo->nRes; i++)
        printf("-15.15s %s\n",
               lsInfo->resTable[i].name, lsInfo->resTable[i].des);

    exit(0);
}
```

Note

The returned data structure of every LSLIB function is dynamically allocated inside LSLIB. This storage is automatically freed by LSLIB and re-allocated next time the same LSLIB function is called. An application should never attempt to free the storage returned by LSLIB. If you need to keep this information across calls, make your own copy of the data structure. This applies to all LSLIB function calls.

The above program will produce output similar to the following:

```
% a.out
My cluster name is <test_cluster>
Master host is <hostA>

RESOURCE_NAME      DESCRIPTION
```

r15s	15-second CPU run queue length
r1m	1-minute CPU run queue length (alias: cpu)
r15m	15-minute CPU run queue length
ut	1-minute CPU utilization (0.0 to 1.0)
pg	Paging rate (pages/second)
io	Disk IO rate (Kbytes/second)
ls	Number of login sessions (alias: login)
it	Idle time (minutes) (alias: idle)
tmp	Disk space in /tmp (Mbytes)
swp	Available swap space (Mbytes) (alias: swap)
mem	Available memory (Mbytes)
ncpus	Number of CPUs
ndisks	Number of local disks
maxmem	Maximum memory (Mbytes)
maxswp	Maximum swap space (Mbytes)
maxtmp	Maximum /tmp space (Mbytes)
cpuf	CPU factor
type	Host type
model	Host model
status	Host status
rexpri	Remote execution priority
server	LSF server host
sparc	SUN SPARC
hppa	HPPA architecture
bsd	BSD UNIX
sysv	System V UNIX
hpux	HP-UX UNIX
solaris	SUN SOLARIS
cs	Compute server
fddi	Hosts connected to the FDDI
alpha	DEC alpha

Getting Host Configuration Information

Host configuration information describes the static attributes of individual hosts in the LSF cluster. Examples of such attributes are host type, host model, number of CPUs, total physical memory, and the special resources associated with the host. These attributes are either read from the LSF configuration file, or found out by LIM on starting up.

The host configuration information can be obtained by calling the following LSLIB function:

`struct hostInfo *ls_gethostinfo(resreq, numhosts, hostlist, listsize, options)`

The following parameters are used by this function:

`char *resreq;`	Resource requirements that a host of interest must satisfy
`int *numhosts;`	If `numhosts` is not `NULL`, `*numhosts` contains the size of the returned array
`char **hostlist;`	An array of candidate hosts
`int listsize;`	Number of candidate hosts
`int options;`	Options, currently only `DFT_FROMTYPE`

On success, this function returns an array containing a `hostInfo` structure for each host of interest. On failure, it returns `NULL` and sets `lserrno` to indicate the error.

The `hostInfo` structure is defined in `lsf.h` as

```
struct hostInfo {
    char  hostName[MAXHOSTNAMELEN]; Host name
    char  *hostType;                Host type
    char  *hostModel;               Host model
    float cpuFactor;                CPU factor of the host’s CPUs
    int   maxCpus;                  Number of CPUs on the host
    int   maxMem;                   Size of physical memory on the host in MB
    int   maxSwap;                  Amount of swap space on the host in MB
    int   maxTmp                    Size of the /tmp file system on the host in MB
    int   nDisk;                    Number of disks on the host
    int   nRes;                     Size of the resources array
    char  **resources;              An array of resources configured for the host
    char  *windows;                 Run windows of the host
    int   numIndx;                  Size of the busyThreshold array
    float *busyThreshold;           Array of load thresholds for determining if the host is
                                    busy
    char  isServer;                 TRUE if the host is a server, FALSE otherwise
    char  licensed;                 TRUE if the host has an LSF license, FALSE otherwise
    int   rexPriority;              Default priority for remote tasks execution on the host
};
```

Note

> *On Solaris, when referencing* `MAXHOSTNAMELEN`, `netdb.h` *must be included before* `lsf.h` *or* `lsbatch.h`.

The following example shows how to use the above LSLIB function in a program. This example program displays the name, host type, total memory, number of CPUs and special resources for each host that has more than 50MB of total memory.

```
#include <netdb.h>    /* Required for Solaris to reference MAXHOSTNAMELEN */
#include <lsf/lsf.h>
#include <stdio.h>

main()
{
    struct hostInfo *hostinfo;
    char   *resreq;
    int    numhosts = 0;
    int    options = 0;
    int    i, j;

    resreq = “maxmem>50”;
    hostinfo = ls_gethostinfo(resreq, &numhosts, NULL, 0, options);

    if (hostinfo == NULL) {
        ls_perror(“ls_gethostinfo”);
        exit(-10);
    }

    printf(“There are %d hosts with more than 50MB total memory\n\n”,
                numhosts);
    printf(“%-11.11s %8.8s %6.6s %6.6s %9.9s\n”,
           “HOST_NAME”, “type”, “maxMem”, “ncpus”, “RESOURCES”);

    for (i = 0; i < numhosts; i++) {
        printf(“%-11.11s %8.8s %8.0fM “, hostinfo[i].hostName,
                hostinfo[i].hostType);

        if (hostinfo[i].maxMem > 0)
            printf(“%6d “, hostinfo[i].maxMem);
        else                        /* maxMem info not available for this host*/
            printf(“%6.6s “, “-”);

        if (hostinfo[i].maxCpus > 0)
            printf(“%6d “, hostinfo[i].maxCpus);
        else                        /* ncpus is not known for this host*/
            printf(“%6.6s“, “-”);

        for (j = 0; j < hostinfo[i].nRes; j++)
```

```
            printf(" %s", hostinfo[i].resources[j]);

        printf("\n");
    }
    exit(0);
}
```

In the above example, `resreq` is the resource requirements used to select the hosts. The variables you can use in a resource requirements must be the resource names returned from `ls_info()`. You can also run the `lsinfo` command to obtain a list of valid resource names in your LSF cluster.

Note that `NULL` and `0` were supplied for the third and fourth parameters of the `ls_gethostinfo()` call. This causes all LSF hosts meeting `resreq` to be returned. If a host list parameter is supplied with this call, the selection of hosts will be limited to those belonging to the list.

If `resreq` is `NULL`, then the default resource requirements will be used. See 'Handling Default Resource Requirements' on page 26 for details.

Note the test of `maxMem` and `maxCpus`. The values of these fields (along with `maxSwap`, `maxTmp` and `nDisks`) are determined when LIM starts on a host. If the host is unavailable, the master LIM supplies a negative value.

The above example program produces output similar to the following:

```
% a.out
There are 4 hosts with more than 50MB total memory

HOST_NAME   type     maxMem ncpus RESOURCES
hostA      HPPA10    128M    1    hppa hpux cs
hostB      ALPHA      58M    2    alpha cs
hostD      ALPHA      72M    4    alpha fddi
hostC      SUNSOL     54M    1    solaris fddi
```

LSLIB also provides functions simpler than `ls_gethostinfo()` to get frequently used information. These functions include:

```
char  *ls_gethosttype(hostname)
char  *ls_gethostmodel(hostname)
float *ls_gethostfactor(hostname)
```

See 'List of LSF API Functions' on page 99 for more details about these functions.

Handling Default Resource Requirements

Some LSLIB functions require a resource requirement parameter. This parameter is passed to LIM for host selection. It is important to understand how LSF handles default resource requirements. See the *LSF User's Guide* for further information about resource requirements.

It is desirable that LSF automatically assume default values for some key requirements if they are not specified by the user.

The default resource requirements depend on the specific application context. For example, the `lsload` command would assume '`type==any order[r15s:pg]`' as the default resource requirements, while `lsrun` assumes '`type==local order[r15s:pg]`' as the default resource requirements. This is because the user usually expects `lsload` to show the load on all hosts, while, with `lsrun`, a conservative approach of running task on the same host type as the local host will in most cases cause the task to be run on the correct host type.

LSLIB provides flexibility for the application programmer to decide what the default behavior should be.

LSF default resource requirements contain two parts, a *type requirement* and an *order requirement*. The former makes sure that the correct type of hosts are selected, while the latter is used to order the selected hosts according to some reasonable criteria.

LSF appends a *type resource requirement* to the resource requirement string supplied by an application in the following situations:

- `resreq` is `NULL` or an empty string.
- `resreq` does not contain a *boolean resource*, for example, '`hppa`', and does not contain a *`type`* or *`model` resource*, for example, '`type==solaris`', '`model==HP715`'.

The default type requirement can be either '`type==any`' or '`type==$fromtype`' depending on whether or not the flag `DFT_FROMTYPE` is set in the `options` parameter of the function call, where `DFT_FROMTYPE` is defined in `lsf.h`.

If DFT_FROMTYPE is set in the options parameter, the default *type requirement is* 'type==$fromtype'. If DFT_FROMTYPE is not set, then the default *type requirement* is 'type==any'.

The value of fromtype depends on the function call. If the function has a fromhost parameter, then fromtype is the host type of the fromhost. Otherwise, fromtype is 'local'.

LSF also appends an *order requirement*, order[r15s:pg], to the resource requirement string if an *order requirement* is not already specified.

The table below lists some examples of how LSF appends the default resource requirements.

Table 2. Examples of Default Resource Requirements

User's Resource Requirement	Resource Requirement After Appending the Default	
	DFT_FROMTYPE set	DFT_FROMTYPE not set
NULL	type==$fromtype order[r15s:pg]	type==any order[r15s:pg]
hpux	hpux order[r15s:pg]	hpux order[r15s:pg]
order[r1m]	type==$fromtype order[r1m]	type==any order[r1m]
model==hp735	model==hp735 order[r15s:pg]	model==hp735 order[r15s:pg]
sparc order[ls]	sparc order[ls]	sparc order[ls]
swp>25 && it>10	swp>25 && it>10 && type==$fromtype order[r15s:pg]	swp>25 && it>10 && type==any order[r15s:pg]
ncpus>1 order[ut]	ncpus>1 && type==$fromtype order[ut]	ncpus>1 && type==any order[ut]

Getting Dynamic Load Information

LSLIB provides several functions to obtain dynamic load information about hosts. The dynamic load information is updated periodically by LIM. The definition of all resources is stored in the `struct lsInfo` data structure returned by the `ls_info(3)` API call (see ‘Getting General Cluster Configuration Information’ on page 19 for details). We can classify LSF resources into two groups by resource location, namely host-based resources and shared resources (see Chapter 2 of the *LSF Batch Administrator’s Guide* for more information on host-based and shared resources).

Getting Dynamic Host-Based Resource Information

Dynamic host-based resources are frequently referred to as load indices, consisting of 11 built-in load indices and a number of external load indices. The built-in load indices report load situation about the CPU, memory, disk subsystem, interactive activities, etc. on each host. The external load indices are optionally defined by your LSF administrator to collect additional host-based dynamic load information that is of interest to your site. The LSLIB function that reports information about load indices is:

`struct hostLoad *ls_load(`*`resreq`*`, `*`numhosts`*`, `*`options`*`, `*`fromhost`*`)`

On success, this function returns an array containing a `hostLoad` structure for each host of interest. On failure, it returns `NULL` and sets `lserrno` to indicate the error.

This function has the following parameters:

`char *resreq;`	Resource requirements that each host of interest must satisfy
`int *numhosts;`	`*numhosts` initially contains the number of hosts requested
`int options;`	Option flags that affect the selection of hosts
`char *fromhost;`	Used in conjunction with the DFT_FROMTYPE option

The value of `*numhosts` determines how many hosts should be returned by this call. If `*numhosts` is 0, information is requested on all hosts satisfying `resreq`. If `numhosts` is `NULL`, load information is requested on one host. If `numhosts` is not `NULL`, then on a successful return `*numhosts` will contain the number of `hostLoad` structures returned.

The options argument is constructed from the bitwise inclusive OR of zero or more of the option flags defined in <lsf/lsf.h>. The most commonly used flags are:

EXACT Exactly *numhosts hosts are desired. If EXACT is set, either exactly *numhosts hosts are returned, or the call returns an error. If EXACT is not set, then up to *numhosts hosts are returned. If *numhosts is zero, then the EXACT flag is ignored and as many hosts in the load sharing system as are eligible (that is, those that satisfy the resource requirement) are returned.

OK_ONLY
Return only those hosts that are currently in the ok state. If OK_ONLY is set, those hosts that are busy, locked, unlicensed or unavail are not returned. If OK_ONLY is not set, then some or all of the hosts whose status are not ok may also be returned, depending on the value of *numhosts and whether the EXACT flag is set.

NORMALIZE
Normalize CPU load indices. If NORMALIZE is set, then the CPU run queue length load indices r15s, r1m, and r15m of each host returned are normalized. See the *LSF User's Guide* for different types of run queue lengths. The default is to return the *raw run queue length*.

EFFECTIVE
If EFFECTIVE is set, then the CPU run queue length load indices of each host returned are the effective load. The default is to return the *raw run queue length*. The options EFFECTIVE and NORMALIZE are mutually exclusive.

DFT_FROMTYPE
This flag determines the default resource requirements. See 'Handling Default Resource Requirements' on page 26 for details.

The fromhost parameter is used when DFT_FROMTYPE is set in options. If fromhost is NULL, the local host is assumed.

`ls_load()` returns an array of the following data structure as defined in `<lsf/lsf.h>`:

```
struct hostLoad {
    char  hostName[MAXHOSTNAMELEN];        Name of the host
    int   status[2];                       The operational and load status of the host
    float *li;                             Values for all load indices of this host
}:
```

The returned `hostLoad` array is ordered according to the *order requirement* in the resource requirements. For details about the ordering of hosts, see the *LSF User's Guide*.

The following example takes no option, and periodically displays the host name, host status and 1-minute effective CPU run queue length for each Sun SPARC host in the LSF cluster.

```
#include <stdio.h>
#include <lsf/lsf.h>

main()
{
    int i;
    struct hostLoad *hosts;
    char   *resreq = “type==sparc”;
    int    numhosts = 0;
    int    options = EFFECTIVE;
    char   *fromhost = NULL;
    char   field[20] = “*”;

    for (;;) {                  /* repeatedly display load */
        hosts = ls_load(resreq, &numhosts, options, fromhost);

        if (hosts == NULL) {
            ls_perror(“ls_load”);
            exit(-1);
        }

        printf(“%-15.15s %6.6s%6.6s\n”, “HOST_NAME”, “status”, “r1m”);

        for (i = 0; i < numhosts; i++) {
            printf(“%-15.15s “, hosts[i].hostName);
            if (LS_ISUNAVAIL(hosts[i].status)) {
                 printf(“%6s\n”, “unavail”);
```

```
            else if (LS_ISBUSY(hosts[i].status))
                printf("%6.6s", "busy");
                else if (LS_ISLOCKED(hosts[i].status))
                printf("%6.6s", "locked");
            else
                 printf("%6.6s", "ok");

            if (hosts[i].li[R1M] >= INFINIT_LOAD)
                printf("%6.6s\n", "-");
            else {
                sprintf(field + 1, "%5.1f", hosts[i].li[R1M]);
                if (LS_ISBUSYON(hosts[i].status, R1M))
                    printf("%6.6s\n", field);
                else
                    printf("%6.6s\n", field + 1);
             }
        }
        sleep(60);             /* until next minute */
    }
}
```

The output of the above program is similar to the following:

```
% a.out
HOST_NAME       status    r1m
hostB             ok      0.0
hostC             ok      1.2
hostA            busy     0.6
hostD            busy     *4.3
hostF            unavail
```

If the host status is `busy` because of `r1m`, then a '*' is printed in front of the value of the `r1m` load index.

In the above example, note that the returned data structure `hostLoad` never needs to be freed by the program even if `ls_load()` is called repeatedly.

Each element of the `li` array is a floating point number between 0.0 and `INFINIT_LOAD` (defined in `lsf.h`). The index value is set to `INFINIT_LOAD` by LSF to indicate an invalid or unknown value for an index.

The `li` array can be indexed using different ways. The constants defined in `lsf.h` (see the `ls_load(3)` man page) can be used to index any built-in load indices as shown in the above example. If external load indices are to be used, the order in which load indices are returned will be the same as that of the resources returned by `ls_info()`. The variables `numUsrIndx` and `numIndx` in structure `lsInfo` can be used to determine which resources are load indices. See 'Advanced Programming Topics' on page 83 for a discussion of more flexible ways to map load index names to values.

LSF defines a set of macros in `lsf.h` to test the `status` field. The most commonly used macros include:

`LS_ISUNAVAIL(status)`
: The LIM on the host is unavailable.

`LS_ISBUSY(status)`
: Returns 1 if the host is busy.

`LS_ISBUSYON(status, index)`
: Returns 1 if the host is busy on the given index.

`LS_ISLOCKED(status)`
: Returns 1 if the host is locked.

`LS_ISOK(status)`
: Returns 1 if none of the above is true.

Getting Dynamic Shared Resource Information

Unlike host-based resources which are inherent properties contributing to the making of each host, shared resources are shared among a set of hosts. The availability of a shared resource is characterized by having multiple instances, with each instance being shared among a set of hosts.

The LSLIB function that can be used to access share resource information is:

```
LS_SHARED_RESOURCE_INFO_T
*ls_sharedresourceinfo(resources, numresources, hostname, options)
```

On success, this function returns an array containing a shared resource information structure (`LS_SHARED_RESOURCE_INFO_T`) for each shared resource. On failure, this function returns NULL and sets lserrno to indicate the error. This function has the following parameters:

```
char **resources;        NULL terminated array of resource names
int  *numresources;      Number of shared resources
int  hostName;           Host name
int  options;            Options (Currently set to 0)
```

resources is a list (NULL terminated array) of shared resource names whose resource information is to be returned. Specify NULL to return resource information for all shared resources defined in the cluster.

numresources is an integer specifying the number of resource information structures (LS_SHARED_RESOURCE_INFO_T) to return. Specify 0 to return resource information for all shared resources in the cluster. On success, numresources is assigned the number of LS_SHARED_RESOURCE_INFO_T structures returned.

hostName is the integer name of a host. Specifying hostName indicates that only the shared resource information for the named host is to be returned. Specify NULL to return resource information for all shared resources defined in the cluster.

ls_sharedresourceinfo returns an array of the following data structure as defined in <lsf/lsf.h> :

```
typedef struct lsSharedResourceInfo {
    char                        *resourceName;   Resource name
    int                         nInstances;      Number of instances
    LS_SHARED_RESOURCE_INST_T   *instances;      pointer to the next instance
} LS_SHARED_RESOURCE_INFO_T;
```

For each shared resource, LS_SHARED_RESOURCE_INFO_T encapsulates an array of instances in the instances field. Each instance is represented by the data type LS_SHARED_RESOURCE_INST_T defined in <lsf/lsf.h> :

```
typedef struct lsSharedResourceInstance {
    char *value;           Value associated with the instance
    int  nHosts;           Number of hosts sharing the instance
    char **hostList;       Hosts associated with the instance
} LS_SHARED_RESOURCE_INST_T;
```

The value field of the LS_SHARED_RESOURCE_INST_T structure contains the ASCII representation of the actual value of the resource. The interpretation of the value requires the knowledge of the resource (Boolean, Numeric or String), which can be obtained from the `resItem` structure accessible through the `lsLoad` structure

returned by `ls_load()`. See 'Getting General Cluster Configuration Information' on page 19 for details.

The following example shows how to use `ls_sharedresourceinfo()` to collect dynamic shared resource information in an LSF cluster. This example displays information from all the dynamic shared resources in the cluster. For each resource, the resource name, instance number, value and locations are displayed.

```
#include <stdio.h>
#include <lsf/lsf.h>
static struct resItem * getResourceDef(char *);
static struct lsInfo  * lsInfo;

void
main()
{
    struct lsSharedResourceInfo *resLocInfo;
    int numRes = 0;
    int i, j, k;

    lsInfo = ls_info();
    if (lsInfo == NULL) {
        ls_perror("ls_info");
        exit(-10);
    }

    resLocInfo = ls_sharedresourceinfo (NULL, &numRes, NULL, 0);

    if (resLocInfo == NULL) {
        ls_perror("ls_sharedresourceinfo");
        exit(-1);
    }

    printf("%-11.11s %8.8s %6.6s %14.14s\n",
            "NAME", "INSTANCE", "VALUE", "LOCATIONS");

    for (k = 0; k < numRes; k++) {
        struct resItem *resDef;
        resDef = getResourceDef(resLocInfo[k].resourceName);
        if (! (resDef->flags & RESF_DYNAMIC))
```

```
            continue;

        printf("%-11.11s", resLocInfo[k].resourceName);
        for (i = 0; i < resLocInfo[k].nInstances; i++) {
            struct lsSharedResourceInstance *instance;

            if (i == 0)
                printf(" %8.1d", i+1);
            else
                printf(" %19.1d", i+1);

            instance = &resLocInfo[k].instances[i];
            printf(" %6.6s", instance->value);

            for (j = 0; j < instance->nHosts; j++)
                if (j == 0)
                    printf(" %14.14s\n", instance->hostList[j]);
                else
                    printf(" %41.41s\n", instance->hostList[j]);

        } /* for */
    } /* for */
} /* main */

static struct resItem *
getResourceDef(char *resourceName)
{
    int i;

    for (i = 0; i < lsInfo->nRes; i++) {
        if (strcmp(resourceName, lsInfo->resTable[i].name) == 0)
            return &lsInfo->resTable[i];
    }

    /* Fail to find the matching resource */
    fprintf(stderr, "Cannot find resource definition for <%s>\n",
            resourceName);

    exit (-1);

}
```

The output of the above program is similar to the following:

```
% a.out
NAME        INSTANCE  VALUE    LOCATIONS
dynamic1           1      2        hostA
                                   hostC
                                   hostD
                   2      4        hostB
                                   hostE
dynamic2           1      3        hostA
                                   hostE
```

Note that the resource `dynamic1` has two instances, one contains two resource units shared by `hostA`, `hostC` and `hostD` and the other contains four resource units shared by `hostB` and `hostE`. The dynamic2 resource has only one instance with three resource units shared by `hostA` and `hostE`.

Making a Placement Decision

If you are writing an application that needs to run tasks on the best available hosts, you need to make *placement decision* as to on which host each task should run.

Placement decision takes two factors into consideration. The first factor is the resource requirements of the task. Every task has a certain set of resource requirements. These may be static, such as a particular hardware architecture or operating system, or dynamic, such as a certain amount of swap space for virtual memory.

LSLIB provides services for placement advice. All you have to do is to call the appropriate LSLIB function with appropriate resource requirements.

A placement advice can be obtained by calling either `ls_load()` function or `ls_placereq()` function. `ls_load()` returns a placement advice together with load index values. `ls_placereq()` returns only the qualified host names. The result list of hosts are ordered by preference, with the first being the best. `ls_placereq()` is useful when a simple placement decision would suffice. `ls_load()` can be used if the placement advice from LSF must be adjusted by your additional criteria. The LSF utilities `lsrun`, `lsmake`, `lslogin`, and `lstcsh` all use `ls_placereq()` for placement decision, whereas lsbatch uses `ls_load()` to get an ordered list of qualified hosts, and then makes placement decisions by considering lsbatch-specific policies.

In order to make optimal placement decisions, it is important that your resource requirements best describe the resource needs of the application. For example, if your task is memory intensive, then your resource requirement string should have '`mem`' in the `order` segment, '`fddi order[mem:r1m]`'.

The LSLIB function, `ls_placereq()`, takes the form of

```
char **ls_placereq(resreq, num, options, fromhost)
```

On success, this function returns an array of host names that best meet the resource requirements. Hosts may be duplicated for hosts that have sufficient resources to accept multiple tasks (for example, multiprocessors).

On failure, this function returns `NULL` and sets `lserrno` to indicate the error.

The parameters for `ls_placereq()` are very similar to those of the `ls_load()` function described in the previous section.

LSLIB will append default resource requirement to resreq according to the rules described in 'Handling Default Resource Requirements' on page 26.

Preference is given to `fromhost` over remote hosts that do not have significantly lighter load or greater resources. This preference avoids unnecessary task transfer and reduces overhead. If `fromhost` is `NULL`, then the local host is assumed.

The example program below takes a resource requirement string as an argument and displays the host in the LSF cluster that best satisfies the resource requirement.

```
#include <stdio.h>
#include <lsf/lsf.h>

main(argc, argv)
    int  argc;
    char *argv[];
{
    char *resreq = argv[1];
    char **best;
    int  num = 1;
```

```
    int  options = 0;
    char *fromhost = NULL;

    if (argc != 2 ) {
        fprintf(stderr, “Usage: %s resreq\n”, argv[0]);
        exit(-2);
    }

    best = ls_placereq(resreq, &num, options, fromhost);
    if (best == NULL) {
        ls_perror(“ls_placereq()”);
        exit(-1);
    }
    printf(“The best host is <%s>\n”, best[0]);

    exit(0);
}
```

The above program will produce output similar to the following:

```
% a.out “type==local order[r1m:ls]”
The best host is <hostD>
```

LSLIB also provides a variant of `ls_placereq()`. `ls_placeofhosts()` lets you provide a list of candidate hosts. See the `ls_policy(3)` man page for details.

Getting Task Resource Requirements

Host selection relies on resource requirements. To avoid the need to specify resource requirements each time you execute a task, LSF maintains a list of task names together with their default resource requirements for each user. This information is kept in three task list files: the system-wide defaults, the per-cluster defaults, and the per-user defaults.

A user can put a task name together with its resource requirements into his / her remote task list by running the `lsrtasks` command. The `lsrtasks` command can be used to add, delete, modify, or display a task entry in the task list. For more information on remote task list and an explanation of resource requirement strings, see the *LSF User's Guide*.

LSLIB provides a function to get the resource requirements associated with a task name. With this function, LSF applications or utilities can automatically retrieve the resource requirements of a given task if the user does not explicitly specify it. For example, the LSF utility `lsrun` tries to find the resource requirements of the user-typed command automatically if '`-R`' option is not specified by the user on the command line.

The LSLIB function call `ls_resreq()` obtains resource requirements of a given task. The syntax of this function is:

`char *ls_resreq(`*`taskname`*`)`

If `taskname` does not appear in the remote task list, this function returns `NULL`.

Typically the resource requirements of a task are then used for host selection purpose. The following program takes the input argument as a task name, get the associated resource requirements from the remote task list, and then supply the resource requirements to a `ls_placereq()` call to get the best host for running this task.

```
#include <stdio.h>
#include <lsf/lsf.h>

main(argc, argv)
    int  argc;
    char *argv[];
{
    char *taskname = argv[1];
    char *resreq;
    char **best;

    if (argc != 2 ) {
        fprintf(stderr, “Usage: %s taskname\n”, argv[0]);
        exit(-1);
    }

    resreq = ls_resreq(taskname);

    if (resreq)
        printf(“Resource requirement for %s is \”%s\”.\n”, taskname, resreq);
    else
        printf(“Resource requirement for %s is NULL.\n”, taksname);
```

```
    best = ls_placereq(resreq, NULL, 0, NULL);
    if (best == NULL) {
        ls_perror("ls_placereq");
        exit(-1);
    }
    printf("Best host for %s is <%s>\n", taskname, best[0]);

    exit(0);
}
```

The above program will produce output similar to the following:

```
% a.out myjob
Resource requirement for myjob is "swp>50 order[cpu:mem]"
Best host for myjob is <hostD>
```

Using Remote Execution Services

Remote execution of interactive tasks in LSF is supported through the Remote Execution Server (RES). The RES listens on a well-known port for service requests. Applications initiate remote execution by making an LSLIB call.

Remote Execution Mechanisms

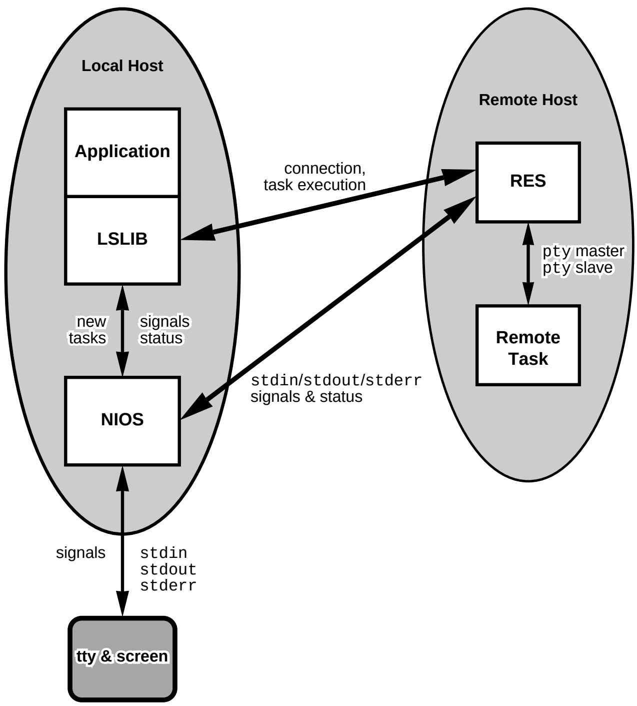
The following steps are typically involved during a remote execution:

- The application makes a remote execution request through LSLIB.
- The LSLIB establishes a connection with the RES on the remote host and passes the client's identity and current execution environment over to the RES.
- The LSLIB starts a Network I/O Server (NIOS) locally if one has not been started already and waits for a call back from the RES.
- If the LSLIB remote execution function is called with the pseudo-terminal option, the RES creates a pseudo-terminal for the remote task and calls back to the client's NIOS to establish terminal I/O channels. If a pseudo-terminal is not required, the RES creates a socket pair instead.

- The RES forks and executes the remote task with its `stdin`, `stdout` and `stderr` associated with the pseudo-terminal or socket. The remote task runs, and the RES forwards any output from the remote task back to the client's NIOS.

- The client's NIOS forwards the output from the remote task to the client's stdout or stderr. The NIOS also watches the user's terminal and forwards any input to the remote task through the RES. Signals received by the NIOS also are forwarded to the remote task.

Figure 5. Remote Execution Mechanisms



When the remote task finishes, the RES collects its status and resource usage and sends them back to the client through its NIOS

Note that all of the above transactions are triggered by an LSLIB remote execution function call and take place transparently to the programmer. Figure 5 shows the relationships between these entities.

The same NIOS is shared by all remote tasks running on different hosts started by the same instance of LSLIB. The LSLIB contacts multiple RESes and they all call back to the same NIOS. The sharing of the NIOS is restricted to within the same application.

Remotely executed tasks behave as if they were executing locally. The local execution environment passed to the RES is re-established on the remote host, and the task's status and resource usage are passed back to the client. Terminal I/O is transparent, so even applications such as `vi` that do complicated terminal manipulation run transparently on remote hosts. UNIX signals are supported across machines, so that remote tasks get signals as if they were running locally. Job control also is done transparently. This level of transparency is maintained between heterogeneous hosts.

Initializing an Application for Remote Execution

Before executing a task remotely, an application must call the following LSLIB function:

```
int ls_initrex(numports, options)
```

On success, this function initializes the LSLIB for remote execution. If your application is installed as a setuid program, this function returns the number of socket descriptors bound to privileged ports. If your program is not installed as a setuid to root program, this function returns `numports` on success.

On failure, this function returns -1 and sets the global variable `lserrno` to indicate the error.

Note

This function must be called before any other remote execution function (see `ls_rex(3)`*) or any remote file operation function (see* `ls_rfs(3)`*) in LSLIB can be called.*

`ls_initrex()` has the following parameters:

`int`	`numports;`	The number of priviliged ports to create
`int`	`options;`	either KEEPUID or `0`

If your program is installed as a setuid to root program, `numports` file descriptors, starting from `FIRST_RES_SOCK` (defined in <`lsf/lsf.h`>), are bound to privileged ports by `ls_initrex()`. These sockets are used only for remote connections to RES. If `numports` is 0, then the system will use the default value `LSF_DEFAULT_SOCKS` defined in `lsf.h`.

By default, `ls_initrex()` restores the effective user ID to real user ID if the program is installed as a setuid to root program. If `options` is `KEEPUID` (defined in `lsf.h`), `ls_initrex()` preserves the current effective user ID. This option is useful if the application needs to be a setuid to root program for some other purpose as well and does not want to go back to real user ID immediately after `ls_initrex()`.

CAUTION!

> **If `KEEPUID` flag is set in options, you must make sure that your application restores back to the real user ID at a proper time of the program execution.**

`ls_initrex()` function selects the security option according to the following rule: if the application program invoking it has an effective uid of root, then privileged ports are created; otherwise, no such port is created and, at remote task start-up time, RES will use the authentication protocol defined by `LSF_AUTH` in the `lsf.conf` file.

Running a Task Remotely

The example program below runs a command on one of the best available hosts. It makes use of the `ls_resreq()` function described in 'Getting Task Resource Requirements' on page 38, the `ls_placereq()` function described in 'Making a Placement Decision' on page 36, the `ls_initrex()` function described in 'Initializing an Application for Remote Execution' on page 42, and the following LSLIB function:

`int ls_rexecv(`*`host`*`, `*`argv`*`, `*`options`*`)`

This function executes a program on the specified host. It does not return if successful. It returns -1 on failure.

This function is basically a remote `execvp`. If a connection with the RES on *host* has not been established, `ls_rexecv()` sets one up. The remote execution environment is set up to be exactly the same as the local one and is cached by the remote RES server. This LSLIB function has the following parameters:

char	*host;	The execution host
char	*argv[];	The command and its arguments
int	options;	See below

The `options` argument is constructed from the bitwise inclusive OR of zero or more of the option flags defined in `<lsf/lsf.h>` with names starting with ‘`REXF_`”. The most commonly used flag is:

`REXF_USEPTY`

Use a remote pseudo terminal as the `stdin`, `stdout`, and `stderr` of the remote task. This option provides a higher degree of terminal I/O transparency. This is only necessary for executing interactive screen applications such as `vi`. The use of a pseudo-terminal incurs more overhead and should be used only if necessary.

LSLIB also provides an `ls_rexecve(3)` function that allows you to specify the environment to be set up on the remote host.

The program follows:

```
#include <stdio.h>
#include <lsf/lsf.h>

main(argc, argv)
    int  argc;
    char *argv[];
{
    char *command = argv[1];
    char *resreq;
    char **best;
    int  num = 1;

    if (argc < 2 ) {
        fprintf(stderr, “Usage: %s command [argument ...]\n”, argv[0]);
        exit(-1);
    }

    if (ls_initrex(1, 0) < 0) {
        ls_perror(“ls_initrex”);
```

```
        exit(-1);
    }

    resreq = ls_resreq(command);

    best = ls_placereq(resreq, &num, 0, NULL);
    if (host == NULL) {
        ls_perror("ls_placereq()");
        exit(-1);
    }

    printf("<<Execute %s on %s>>\n", command, best[0]);
    ls_rexecv(best[0], argv + 1, 0);
    /* should never get here */
    ls_perror("ls_rexecv()");
    exit(-1);
}
```

The output of the above program would be something like:

```
% a.out myjob
<<Execute myjob on hostD>>
(output from myjob goes here ....)
```

Note

Any application that uses LSF's remote execution service must be installed for proper authentication. See 'Authentication' on page 17.

The LSF utility `lsrun` is implemented using the `ls_rexecv()` function. After remote task is initiated, `lsrun` calls the `ls_rexecv()` function, which then executes NIOS to handle all input/output to and from the remote task and exits with the same status when remote task exits.

See 'Advanced Programming Topics' on page 83 for an alternative way to start remote tasks.

3 Programming with LSBLIB

This chapter shows how to use LSBLIB to access the services provided by LSF Batch and LSF JobScheduler. Since LSF Batch and LSF JobScheduler are built on top of LSF Base, LSBLIB relies on services provided by LSLIB. Thus if you use LSBLIB functions, you must link your program with both LSLIB and LSBLIB.

LSF Batch and LSF JobScheduler services are mostly provided by `mbatchd`, except services for processing event and job log files which do not involve any daemons. LSBLIB is shared by both LSF Batch and LSF JobScheduler. The functions described for LSF Batch in this chapter also apply to LSF JobScheduler, unless explicitly indicated otherwise.

Initializing LSF Batch Applications

Before accessing any of the services provided by the LSF Batch and LSF JobScheduler, an application must initialize LSBLIB. It does this by calling the following function:

```
int lsb_init(appname);
```

On success, it returns 0; otherwise, it returns -1 and sets `lsberrno` to indicate the error.

The parameter `appname` is used only if you want to log detailed messages about the transactions inside LSLIB for debugging purpose. The messages will be logged only if `LSB_CMD_LOG_MASK` is defined as `LOG_DEBUG1`.

The messages will be logged in file `LSF_LOGDIR/appname`. If *appname* is `NULL`, the log file is `LSF_LOGDIR/bcmd`.

Note

This function must be called before any other function in LSBLIB can be called.

Getting Information about LSF Batch Queues

LSF Batch queues hold the jobs in the LSF Batch and set scheduling policies and limits on resource usage.

LSBLIB provides a function to get information about the queues in the LSF Batch. This includes queue name, parameters, statistics, status, resource limits, scheduling policies and parameters, and users and hosts associated with the queue.

The example program in this section uses the following LSBLIB function to get the queue information:

`struct queueInfoEnt *lsb_queueinfo(`*`queues`*`,`*`numQueues`*`,`*`hostname`*`,`*`username`*`,`*`options`*`)`

On success, this function returns an array containing a `queueInfoEnt` structure (see below) for each queue of interest and sets `*numQueues` to the size of the array. On failure, it returns `NULL` and sets `lsberrno` to indicate the error. It has the following parameters:

```
char   **queues;        An array containing names of queues of interest
int    *numQueues;      The number of names in queues
char   *hostname;       Only queues using hostname are of interest
char   *username;       Only queues enabled for user are of interest
int    options;         Reserved for future use; supply 0
```

To get information on all queues, set `*numQueues` to `0`; `*numQueues` will be updated to the actual number of queues returned on a successful return.

If `*numQueues` is `1` and `queue` is `NULL`, information on the system default queue is returned.

If `hostname` is not `NULL`, then all queues using host `hostname` as a batch server host will be returned. If `username` is not `NULL`, then all queues allowing user `username` to submit jobs to will be returned.

The `queueInfoEnt` structure is defined in `lsbatch.h` as

```
struct queueInfoEnt {
    char   *queue;          Name of the queue
    char   *description;    Description of the queue
```

```
int   priority;            Priority of the queue
short nice;                Nice value at which jobs in the queue will be run
char  *userList;           Users allowed to submit jobs to the queue
char  *hostList;           Hosts to which jobs in the queue may be dispatched
int   nIdx;                Size of the loadSched and loadStop arrays
float *loadSched;          Load thresholds that control scheduling of jobs from the queue
float *loadStop;           Load thresholds that control suspension of jobs from the queue
int   userJobLimit;        Number of unfinished jobs a user can dispatch from the queue
int   procJobLimit;        Number of unfinished jobs the queue can dispatch to a processor
char  *windows;            Queue run window
int   rLimits[LSF_RLIM_NLIMITS];   The per-process resource limits for jobs
char  *hostSpec;           Obsolete. Use defaultHostSpec instead
int   qAttrib;             Attributes of the queue
int   qStatus;             Status of the queue
int   maxJobs;             Job slot limit of the queue.
int   numJobs;             Total number of job slots required by all jobs
int   numPEND;             Number of  job slots needed by pending jobs
int   numRUN;              Number of jobs slots used by  running jobs
int   numSSUSP;            Number of  job slots used by system suspended jobs
int   numUSUSP;            Number of  jobs slots used by user suspended jobs
int   mig;                 Queue migration threshold in minutes
int   schedDelay;          Schedule delay for new jobs
int   acceptIntvl;         Minimum interval between two jobs dispatched to the same host
char  *windowsD;           Queue dispatch window
char  *nqsQueues;          A blank-separated list of NQS queue specifiers
char  *userShares;         A blank-separated list of user shares
char  *defaultHostSpec;  Value of DEFAULT_HOST_SPEC for the queue in lsb.queues
int   procLimit;           Maximum number of job slots a job can take
char  *admins;             Queue level administrators
char  *preCmd;             Queue level pre-exec command
char  *postCmd;            Queue’s post-exec command
char  *requeueEValues;     Queue’s requeue exit status
int   hostJobLimit;        Per host job slot limit
char  *resReq;             Queue level resource requirement
int   numRESERVE;          Reserved job slots for pending jobs
int   slotHoldTime;        Time period for reserving job slots
char  *sndJobsTo;          Remote queues to forward jobs to
char  *rcvJobsFrom;        Remote queues which can forward to me
char  *resumeCond;         Conditions to resume jobs
char  *stopCond;           Conditions to suspend jobs
char  *jobStarter;         Queue level job starter
char  *suspendActCmd;      Action commands for SUSPEND
char  *resumeActCmd;       Action commands for RESUME
char  *terminateActCmd;  Action commands for TERMINATE
```

```
    int     sigMap[LSB_SIG_NUM]; Configurable signal mapping
    char    *preemption;        Preemption policy
    int      maxRschedTime;    Time period for remote cluster to schedule job
};
```

The variable `nIdx` is the number of load threshold values for job scheduling. This is in fact the total number of load indices as returned by LIM. The parameters `sndJobsTo`, `rcvJobsFrom`, and `maxRschedTime` are only used with LSF MultiCluster.

For a complete description of the fields in the `queueInfoEnt` structure, see the `lsb_queueinfo(3)` man page.

The program below takes a queue name as the first argument and displays information about the named queue.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

int
main (argc, argv)
    int  argc;
    char *argv[];
{
    struct queueInfoEnt *qInfo;
    int  numQueues = 1;
    char *queue=argv[1];
    int  i;

    if (argc != 2) {
        printf("Usage: %s queue_name\n", argv[0]);
        exit(-1);
    }

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init()");
        exit(-1);
    }

    qInfo = lsb_queueinfo(&queue, &numQueues, NULL, NULL, 0);
    if (qInfo == NULL) {
        lsb_perror("lsb_queueinfo()");
        exit(-1);
    }
```

```
    printf("Information about %s queue:\n", queue);
    printf("Description: %s\n", qInfo[0].description);
    printf("Priority: %d                     Nice:     %d     \n",
            qInfo[0].priority, qInfo[0].nice);
    printf("Maximum number of job slots:");
    if (qip->maxJobs < INFINIT_INT)
        printf("%5d\n", qInfo[0].maxJobs);
    else
        printf("%5s\n", "unlimited");

    printf("Job slot statistics: PEND(%d) RUN(%d) SUSP(%d) TOTAL(%d).\n",
         qInfo[0].numPEND, qInfo[0].numRUN,
         qInfo[0].numSSUSP + qInfo[0].numUSUSP, qInfo[0].numJobs);

    exit(0);
}
```

The header file `lsbatch.h` must be included with every application that uses LSBLIB functions. Note that `lsf.h` does not have to be explicitly included in your program because `lsbatch.h` already has `lsf.h` included. The function `lsb_perror()` is used in much the same way `ls_perror()` is used to print error messages regarding function call failure. You could check `lsberrno` if you want to take different actions for different errors.

In the above program, `INFINIT_INT` is defined in `lsf.h` and is used to indicate that there is no limit set for `maxJobs`. This applies to all LSF API function calls. LSF will supply `INFINIT_INT` automatically whenever the value for the variable is either invalid (not available) or infinity. This value should be checked for all variables that are optional. For example, if you were to display the `loadSched` / `loadStop` values, an `INFINIT_INT` indicates that the threshold is not configured and is ignored.

Note

Like the returned data structures by LSLIB functions, the returned data structures from an LSBLIB function is dynamically allocated inside LSBLIB and is automatically freed next time the same function is called. You should not attempt to free the space allocated by LSBLIB. If you need to keep this information across calls, make your own copy of the data structure.

The above program will produce output similar to the following:

```
Information about normal queue:
Description: For normal low priority jobs
Priority: 25            Nice: 20
Maximum number of job slots : 40
Job slot statistics: PEND( 5) RUN(12) SUSP(1) TOTAL(18)
```

Getting Information about LSF Batch Hosts

LSF Batch server hosts execute the jobs in the LSF Batch system.

LSBLIB provides a function to get information about the server hosts in the LSF Batch system. This includes both configured static information as well as dynamic information. Examples of host information include host name, status, job limits and statistics, dispatch windows, and scheduling parameters.

The example program in this section uses the following LSBLIB function:

struct hostInfoEnt *lsb_hostinfo(*hosts*, *numHosts*)

This function gets information about LSF Batch server hosts. On success, it returns an array of `hostInfoEnt` structures which hold the host information and sets `*numHosts` to the size of the array. On failure, it returns `NULL` and sets `lsberrno` to indicate the error. It has the following parameters:

char	**hosts;	An array of names of hosts of interest
int	*numHosts;	The number of names in hosts

To get information on all hosts, set `*numHosts` to `0`; `*numHosts` will be set to the actual number of `hostInfoEnt` structures when this call returns successfully.

If `*numHosts` is `1` and `hosts` is `NULL`, information on the local host is returned.

The `hostInfoEnt` structure is defined in `lsbatch.h` as

```
struct hostInfoEnt {
    char  *host;            Name of the host
    int   hStatus;          Status of host. (see below)
```

```
    int   busySched;       Reason host will not schedule jobs
    int   busyStop;        Reason host has suspended jobs
    float cpuFactor;       Host CPU factor, as returned by LIM
    int   nIdx;            Size of the loadSched and loadStop arrays, as returned from LIM
    float *load;           Load LSF Batch used for scheduling batch jobs
    float *loadSched;      Load thresholds that control scheduling of jobs on host
    float *loadStop;       Load thresholds that control suspension of jobs on host
    char  *windows;        Host dispatch window
    int   userJobLimit;    Maximum number of jobs a user can run on host
    int   maxJobs;         Maximum number of jobs that host can process concurrently
    int   numJobs;         Number of jobs running or suspended on host
    int   numRUN;          Number of jobs running on host
    int   numSSUSP;        Number of jobs suspended by sbatchd on host
    int   numUSUSP;        Number of jobs suspended by a user on host
    int   mig;             Migration threshold for jobs on host
    int   attr;            Host attributes
#define H_ATTR_CHKPNTABLE  0x1
#define H_ATTR_CHKPNT_COPY 0x2
    float *realLoad;       The load mbatchd obtained from LIM
    int   numRESERVE;      Num of slots reserved for pending jobs
    int    chkSig;         This variable is obsolete
};
```

Note the differences between host information returned by LSLIB function `ls_gethostinfo()` and host information returned by the LSBLIB function `lsb_hostinfo()`. The former returns general information about the hosts whereas the latter returns LSF Batch specific information about hosts.

For a complete description of the fields in the `hostInfoEnt` structure, see the `lsb_hostinfo(3)` man page.

The example program below takes a host name as an argument and displays various information about the named host. It is a simplified version of the LSF Batch `bhosts` command.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

main (argc, argv)
    int  argc;
    char *argv[];
{
    struct hostInfoEnt *hInfo;
```

```
int  numHosts = 1;
char *hostname = argv[1];
int  i;

if (argc != 2) {
    printf("Usage: %s hostname\n", argv[0]);
    exit(-1);
}
if (lsb_init(argv[0]) < 0) {
    lsb_perror("lsb_init");
    exit(-1);
}

hInfo = lsb_hostinfo(&hostname, &numHosts);

if (hInfo == NULL) {
    lsb_perror("lsb_hostinfo");
    exit (-1);
}

printf("HOST_NAME    STATUS    JL/U  NJOBS  RUN  SSUSP USUSP\n");

printf ("%-18.18s", hInfo->host);

if (hInfo->hStatus & HOST_STAT_UNLICENSED) {
    printf(" %-9s\n", "unlicensed");
    continue;                 /* don't print other info */
} else if (hInfo->hStatus & HOST_STAT_UNAVAIL)
    printf(" %-9s",  "unavail");
else if (hInfo->hStatus & HOST_STAT_UNREACH)
    printf(" %-9s", "unreach");
else if (hInfo->hStatus & ( HOST_STAT_BUSY | HOST_STAT_WIND
        | HOST_STAT_DISABLED | HOST_STAT_LOCKED
        | HOST_STAT_FULL | HOST_STAT_NO_LIM))
    printf(" %-9s", "closed");
else
    printf(" %-9s", "ok");

if (hInfo->userJobLimit < INFINIT_INT)
    printf("%4d", hInfo->userJobLimit);
else
    printf("%4s", "-");

printf("%7d  %4d  %4d  %4d\n",
```

```
        hInfo->numJobs, hInfo->numRUN, hInfo->numSSUSP, hInfo->numUSUSP);

    exit(0);

}
```

`hStatus` is the status of the host. It is the bitwise inclusive `OR` of some of the following constants defined in `lsbatch.h`:

`HOST_STAT_BUSY`
: The host load is greater than a scheduling threshold. In this status, no new batch job will be scheduled to run on this host.

`HOST_STAT_WIND`
: The host dispatch window is closed. In this status, no new batch job will be accepted.

`HOST_STAT_DISABLED`
: The host has been disabled by the LSF administrator and will not accept jobs. In this status, no new batch job will be scheduled to run on this host.

`HOST_STAT_LOCKED`
: The host is locked by an exclusive job. In this status, no new batch job will be scheduled to run on this host.

`HOST_STAT_FULL`
: The host has reached its job limit. In this status, no new batch job will be scheduled to run on this host.

`HOST_STAT_UNREACH`
: The `sbatchd` on this host is unreachable.

`HOST_STAT_UNAVAIL`
: The LIM and `sbatchd` on this host are unreachable.

`HOST_STAT_UNLICENSED`
: The host does not have an LSF license.

`HOST_STAT_NO_LIM`
: The host is running an `sbatchd` but not a LIM.

If none of the above holds, `hStatus` is set to `HOST_STAT_OK` to indicate that the host is ready to accept and run jobs.

The constant `INFINIT_INT` defined in `lsf.h` is used to indicate that there is no limit set for `userJobLimit`.

The example output from the above program follows:

```
% a.out hostB
HOST_NAME    STATUS    JL/U  NJOBS  RUN  SSUSP USUSP
hostB         ok        -     2      1    1     0
```

Job Submission and Modification

Job submission and modification are most common operations in the LSF Batch system. A user can submit jobs to the system and then modify them if the job has not been started.

LSBLIB provides one function for job submission and one function for job modification.

```
int lsb_submit(jobSubReq, jobSubReply)
int lsb_modify(jobSubReq, jobSubReply, jobId)
```

On success, these calls return the job ID, otherwise -1 is returned with `lsberrno` set to indicate the error. These two functions are similar except that `lsb_modify()` modifies the parameters of an already submitted job.

Both of these functions use the same data structure:

```
struct submit      *jobSubReq;       Job specifications
struct submitReply *jobSubReply;     Results of job submission
int   jobId;                         Id of the job to modify (lsb_modify() only)
```

The `submit` structure is defined in `lsbatch.h` as

```
struct submit {
    int     options;          Indicates which optional fields are present
    int     options2;         Indicates which additional fields are present
```

```
    char    *jobName;         Job name (optional)
    char    *queue;           Submit the job to this queue (optional)
    int     numAskedHosts;    Size of askedHosts (optional)
    char    **askedHosts;     An array of names of candidate hosts (optional)
    char    *resReq;          Resource requirements of the job (optional)
    int     rlimits[LSF_RLIM_NLIMITS];
                              Limits on system resource use by all of the job's processes
    char    *hostSpec;        Host model used for scaling rlimits (optional)
    int     numProcessors;    Initial number of processors needed by the job
    char    *dependCond;      Job dependency condition (optional)
    time_t  beginTime;        Dispatch the job on or after beginTime
    time_t  termTime;         Job termination deadline
    int     sigValue;         This variable is obsolete)
    char    *inFile;          Path name of the job's standard input file (optional)
    char    *outFile;         Path name of the job's standard output file (optional)
    char    *errFile;         Path name of the job's standard error output file (optional)
    char    *command;         Command line of the job
    time_t  chkpntPeriod;     Job is checkpointable with this period (optional)
    char    *chkpntDir;       Directory for this job's chk directory (optional)
    int     nxf;              Sze of xf (optional)
    struct xFile *xf;         An array of file transfer specifications (optional)
    char    *preExecCmd;      Job's pre-execution command (optional)
    char    *mailUser;        User E-mail address to which the job's output are mailed (optional)
    int     delOptions;       Bits to be removed from options (lsb_modify() only)
    char    *projectName;     Name of the job's project (optional)
    int     maxNumProcessors;    Requested maximum num of job slots for the job
    char    *loginShell;      Login shell to be used to re-initialize environment
    char    *exceptList;      Lists the exception handlers
};
```

For a complete description of the fields in the `submit` structure, see the `lsb_submit(3)` man page.

The `submitReply` structure is defined in `lsbatch.h` as

```
struct submitReply {
    char    *queue;               The queue name the job was submitted to
    int     badJobId;             dependCond contains badJobId but there is no such job
    char    *badJobName;          dependCond contains badJobName but there is no such job
    int     badReqIndx;           Index of a host or resource limit that caused an error
};
```

The last three variables in the structure `submitReply` are only used when the `lsb_submit()` or `lsb_modify()` function calls fail.

For a complete description of the fields in the `submitReply` structure, see the `lsb_submit(3)` man page.

To submit a new job, all you have to do is to fill out this data structure and then call `lsb_submit()`. The `delOptions` variable is ignored by LSF Batch system for `lsb_submit()` function call.

The example job submission program below takes the job command line as an argument and submits the job to the LSF Batch system. For simplicity, it is assumed that the job command does not have arguments.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

main(argc, argv)
    int  argc;
    char **argv;
{
    struct submit  req;
    struct submitReply  reply;
    int  jobId;
    int  i;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s command\n", argv[0]);
        exit(-1);
    }

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    req.options = 0;
    req.maxNumProcessors = 1;
    req.options2 = 0;
    req.resReq = NULL;

    for (i = 0; i < LSF_RLIM_NLIMITS; i++)
        req.rLimits[i] = DEFAULT_RLIMIT;
```

```
    req.hostSpec = NULL;
    req.numProcessors = 1;
    req.maxNumProcessors = 1;
    req.beginTime = 0;
    req.termTime  = 0;
    req.command = argv[1];
    req.nxf = 0;
    req.delOptions = 0;

    jobId = lsb_submit(&req, &reply);

    if (jobId < 0) {
        switch (lsberrno) {
        case LSBE_QUEUE_USE:
        case LSBE_QUEUE_CLOSED:
            lsb_perror(reply.queue);
            exit(-1);
        default:
            lsb_perror(NULL);
            exit(-1);
        }
    }
    exit(0);
}
```

The `options` field of the `submit` structure is the bitwise inclusive OR of some of the `SUB_*` flags defined in `lsbatch.h`. These flags serve two purposes. Some flags indicate which of the optional fields of the `submit` structure are present. Those that are not present have default values. Other flags indicate submission options. For a description of these flags, see `lsb_submit(3)`.

Since `options` indicate which of the optional fields are meaningful, the programmer does not need to initialize the fields that are not chosen by options. All parameters that are not optional must be initialized properly.

If the `resReq` field of the `submit` structure is `NULL`, LSBLIB will try to obtain resource requirements for `command` from the remote task list (see 'Getting Task Resource Requirements' on page 38). If the task does not appear in the remote task list, then `NULL` is passed to the LSF Batch system. `mbatchd` will then use the default resource requirements with option `DFT_FROMTYPE` bit set when making a LSLIB call for host selection from LIM. See ‘Handling Default Resource Requirements’ on page 26 for more information about default resource requirements.

The constant DEFAULT_RLIMIT defined in lsf.h indicates that there is no limit on a resource.

The constants used to index the `rlimits` array of the `submit` structure is defined in `lsf.h`, and the resource limits currently supported by LSF Batch are listed below.

Table 3. Resource Limits Supported by LSF Batch

Resource Limit	Index in `rlimits` Array
CPU time limit	LSF_RLIMIT_CPU
File size limit	LSF_RLIMIT_FSIZE
Data size limit	LSF_RLIMIT_DATA
Stack size limit	LSF_RLIMIT_STACK
Core file size limit	LSF_RLIMIT_CORE
Resident memory size limit	LSF_RLIMIT_RSS
Number of open files limit	LSF_RLIMIT_OPEN_MAX
Virtual memory limit	LSF_RLIMIT_SWAP
Wall-clock time run limit	LSF_RLIMIT_RUN
Maximum num of processes a job can fork	LSF_RLIMIT_PROCESS

The `hostSpec` field of the `submit` structure specifies the host model to use for scaling `rlimits[LSF_RLIMIT_CPU]` and `rlimits[LSF_RLIMIT_RUN]` (See `lsb_queueinfo(3)`). If `hostSpec` is `NULL`, the local host’s model is assumed.

If the `beginTime` field of the `submit` structure is `0`, start the job as soon as possible.

If the `termTime` field of the `submit` structure is `0`, allow the job to run until it reaches a resource limit.

The above example checks the value of `lsberrno` when `lsb_submit()` fails. Different actions can be taken depending on the type of the error. All possible error numbers are defined in `lsbatch.h`. For example, error number `LSBE_QUEUE_USE` indicates that the user is not authorized to use the queue. The error number `LSBE_QUEUE_CLOSED` indicates that the queue is closed.

Since a queue name was not specified for the job, the job will be submitted to the default queue. The `queue` field of the `submitReply` structure contains the name of the queue to which the job was submitted.

The above program will produce output similar to the following:

```
Job <5602> is submitted to default queue <default>.
```

The output from the job will be mailed to the user because it did not specify a file name for the `outFile` parameter in the `submit` structure.

If you are familiar with the `bsub` command, it may help to know how the fields in the `submit` structure realte to the `bsub` command options. This is provided in the following table.

Table 4. submit fields and bsub options

bsub Option	submit Field	options
-J job_name_spec	jobName	SUB_JOB_NAME
-q queue_name	queue	SUB_QUEUE
-m host_name[+[pref_level]]	askedHosts	SUB_HOST
-n min_proc[,max_proc]	numProcessors, maxNumProcessors	
-R res_req	resReq	SUB_RES_REQ
-c cpu_limit[/host_spec]	rlimits[LSF_RLIMIT_CPU] / hostSpec **	SUB_HOST_SPEC (if host_spec is specified)
-W run_limit[/host_spec]	rlimits[LSF_RLIMIT_RUN] / hostSpec**	SUB_HOST_SPEC (if host_spec is specified)
-F file_limit	rlimits[LSF_RLIMIT_FSIZE]**	
-M mem_limit	rlimits[LSF_RLIMIT_RSS]**	
-D data_limit	rlimits[LSF_RLIMIT_DATA]**	
-S stack_limit	rlimits[LSF_RLIMIT_STACK**	
-C core_limit	rlimits[LSF_RLIMIT_CORE]**	

Table 4. submit fields and bsub options

bsub Option	submit Field	options
-k "chkpnt_dir [chkpnt_period]"	chkpntDir, chkpntPeriod	SUB_CHKPNT_DIR, SUB_CHKPNT_DIR (if chkpntPeriod is specified)
-w depend_cond	dependCond	SUB_DEPEND_COND
-b begin_time	beginTime	
-t term_time	TermTime	
-i in_file	inFile	SUB_IN_FILE
-o out_file	outFile	SUB_OUT_FILE
-e err_file	errFile	SUB_ERR_FILE
-u mail_user	mailUser	SUB_MAIL_USER
-f "lfile op [rfile]"	xf	
-E "pre_exec_command [argument ...]"	preExecCmd	SUB_PRE_EXEC
-L login_shell	loginShell	SUB_LOGIN_SHELL
-P project_name	projectName	SUB_PROJECT_NAME
-G user_group	userGroup	SUB_USER_GROUP
-H		SUB2_HOLD*
-x		SUB_EXCLUSIVE
-r		SUB_RERUNNABLE
-N		SUB_NOTIFY_END
-B		SUB_NOTIFY_BEGIN
-I		SUB_INTERACTIVE
-Ip		SUB_PTY
-Is		SUB_PTY_SHELL
-K		SUB2_BSUB_BLOCK*

Table 4. submit fields and bsub options

bsub Option	submit Field	options
- X "exception_cond([params]):: action"	exceptList	SUB_EXCEPT
-T time_event	timeEvent	SUB_TIME_EVENT

* indicates a bitwise OR mask for options2.
** indicates -1 means undefined

Even if not all options are used, all optional string fields must be initialized to the empty string. For a complete description of the fields in the `submit` structure, see the `lsb_submit(3)` manual page.

To modify an already submitted job, you can fill out a new submit structure to override existing parameters, and use `delOptions` to remove option bits that were previously specified for the job. Essentially, modifying a submitted job is like re-submitting the job. So the same program as above can be used to modify an existing job with minor changes. One additional parameter that must be specified for job modification is the job Id. The parameter `delOptions` can also be set if you want to clear some option bits that were set previously.

Note

All applications that call `lsb_submit()` *and* `lsb_modify()` *are subject to authentication constraints described in 'Authentication' on page 17.*

Getting Information about Batch Jobs

LSBLIB provides functions to get status information about batch jobs. Since the number of jobs in the LSF Batch system could be on the order of many thousands, getting all this information in one message could potentially use a lot of memory space. LSBLIB allows the application to open a stream connection and then read the job records one by one. This way the memory space needed is always the size of one job record.

LSF Batch Job ID

An LSF Batch job ID stored in a 32-bit integer and it consists of two parts: base ID and array index. The base ID is stored in the lower 20 bits whereas the array index in the top 12 bits which are only used when the underlying job is an array job.

LSBLIB provides the following C macros (defined in `lsbatch.h`) for munipulating job IDs:

`LSB_JOBID(base_id, array_index)`	Yield a 32-bit LSF Batch job ID
`LSB_ARRAY_IDX(job_id)`	Yield array index part of the job ID
`LSB_ARRAY_JOBID(job_id)`	Yield the base ID part of the job ID

The function calls used to get job information are:

```
int lsb_openjobinfo(jobId, jobName, user, queue, host, options);
struct jobInfoEnt *lsb_readjobinfo(more);
void lsb_closejobinfo(void);
```

These functions are used to open a job information connection with `mbatchd`, read job records, and then close the job information connection.

`lsb_openjobinfo()` function takes the following arguments:

`int jobId;`	Select job with the given job Id
`char *jobName;`	Select job(s) with the given job name
`char *user;`	Select job(s) submitted by the named user or user group
`char *queue;`	Select job(s) submitted to the named queue
`char *host;`	Select job(s) that are dispatched to the named host
`int options;`	Selection flags constructed from the bits defined in `lsbatch.h`

The options parameter contains additional job selection flags defined in lsbatch.h. These are:

ALL_JOB
: Select jobs matching any status, including unfinished jobs and recently finished jobs. LSF Batch remembers finished jobs within the CLEAN_PERIOD, as defined in the lsb.params file.

CUR_JOB
: Return jobs that have not finished yet.

DONE_JOB
: Return jobs that have finished recently.

PEND_JOB
: Return jobs that are in the pending status.

SUSP_JOB
: Return jobs that are in the suspended status.

LAST_JOB
: Return jobs that are submitted most recently.

JGRP_ARRAY_INFO
: Return job array information.

If options is 0, then the default is CUR_JOB.

lsb_openjobinfo() returns the total number of matching job records in the connection. It returns -1 on failure and sets lsberrno to indicate the error.

lsb_readjobinfo() takes one argument:

int *more;	If not NULL, contains the remaining number of jobs unread

Either this parameter or the return value from the lsb_openjobinfo() can be used to keep track of the number of job records that can be returned from the connection. This parameter is updated each time lsb_readjobinfo() is called.

The `jobInfoEnt` structure returned by `lsb_readjobinfo()` is defined in `lsbatch.h` as:

```
struct jobInfoEnt {
    int      jobId;                  job ID
    char     *user;                  submission user
    /* possible values for the status field */
#define JOB_STAT_PEND       0x01     job is pending
#define JOB_STAT_PSUSP      0x02     job is held
#define JOB_STAT_RUN        0x04     job is running
#define JOB_STAT_SSUSP      0x08     job is suspended by LSF Batch system
#define JOB_STAT_USUSP      0x10     job is suspended by user
#define JOB_STAT_EXIT       0x20     job exited
#define JOB_STAT_DONE       0x40     job is completed successfully
    int      status;
    int      *reasonTb;              pending or suspending reasons
    int      numReasons;             length of reasonTb vector
    int      reasons;                reserved for future use
    int      subreasons;             reserved for future use
    int      jobPid;                 process Id of the job
    time_t   submitTime;             time when the job is submitted
    time_t   reserveTime;            time when job slots are reserved
    time_t   startTime;              time when job is actually started
    time_t   predictedStartTime;     job's predicted start time
    time_t   endTime;                time when the job finishes
    time_t   lastEvent;              last time event
    time_t   nextEvent;              next time event
    int      duration;               duration time (minutes)
    float    cpuTime;                CPU time consumed by the job
    int      umask;                  file mode creation mask for the job
    char     *cwd;                   current working directory where job is submitted
    char     *subHomeDir;            submitting user's home directory
    char     *fromHost;              host from which the job is submitted
    char     **exHosts;              host(s) on which the job executes
    int      numExHosts;             number of execution hosts
    float    cpuFactor;              CPU factor of the first execution host
    int      nIdx;                   number of load indices in the loadSched and
                                     loadStop vector
    float    *loadSched;             stop scheduling new jobs if this threshold
                                     is exceeded
    float    *loadStop;              stop jobs if this threshold is exceeded
    struct   submit submit;          job submission parameters
    int      exitStatus;             exit status
    int      execUid;                user ID under which the job is running
```

```
    char     *execHome;                home directory of the user denoted by execUid
    char     *execCwd;                 current working directory where job is running
    char     *execUsername;            user name corresponds to execUid
    time_t   jRusageUpdateTime;        last time job's resource usage is updated
    struct   jRusage runRusage;        last updated job's resource usage

    /* Possible values for the jType field */
#define     JGRP_NODE_JOB          1   this structure stores a normal batch job
#define     JGRP_NODE_GROUP        2   this structure stores a job group
#define     JGRP_NODE_ARRAY        3   this structure stores a job array
    int      jType;
    char     *parentGroup;             for job group use
    char     *jName;                   job group name: if jType is JGRP_NODE_GROUP
                                       job's name:     otherwise
    /* index into the counter array, only used for job array */
#define      JGRP_COUNT_NJOBS     0    total jobs in the array
#define      JGRP_COUNT_PEND      1    number of pending jobs in the array
#define      JGRP_COUNT_NPSUSP    2    number of held jobs in the array
#define      JGRP_COUNT_NRUN      3    number of running jobs in the array
#define      JGRP_COUNT_NSSUSP    4    number of jobs suspended by the system in the array
#define      JGRP_COUNT_NUSUSP    5    number of jobs suspended by the user in the array
#define      JGRP_COUNT_NEXIT     6    number of exited jobs in the array
#define      JGRP_COUNT_NDONE     7    number of successfully completed jobs
    int      counter[NUM_JGRP_COUNTERS];
};
```

Under LSF Batch, the `jobInfoEnt` can store a job array as well as a non-array batch job, depending on the value of `jType` field, which can be either `JGRP_NODE_JOB` or `JGRP_NODE_ARRAY`.

`lsb_closejobinfo()` should be called after receiving all job records in the connection.

Below is an example of a simplified `bjobs` command. This program displays all pending jobs belonging to all users.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

main()
{
    int  options = PEND_JOB;
    char *user = "all";             /* match jobs for all users */
```

```
    struct jobInfoEnt *job;
    int more;

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    if (lsb_openjobinfo(0, NULL, user, NULL, NULL, options) < 0) {
        lsb_perror("lsb_openjobinfo");
        exit(-1);
    }

    printf("All pending jobs submitted by all users:\n");
    for (;;) {
        job = lsb_readjobinfo(&more);
        if (job == NULL) {
            lsb_perror("lsb_readjobinfo");
            exit(-1);
        }
        /* display the job */
        printf("%s:\nJob <%d> of user <%s>, submitted from host <%s>\n",
                ctime(&job->submitTime), job->jobId, job->user, job->fromHost);

        if (! more)
            break;
    }

    lsb_closejobinfo();
    exit(0);
}
```

If you want to print out the reasons why the job is still pending, you can use the function `lsb_pendreason()`. See `lsb_pendreason(3)` for details.

The above program will produce output similar to the following:

```
All pending jobs submitted by all users:
Mon Mar 1 10:34:04 EST 1996:
Job <123> of user <john>, submitted from host <orange>
Mon Mar 1 11:12:11 EST 1996:
Job <126> of user <john>, submitted from host <orange>
Mon Mar 1 14:11:34 EST 1996:
Job <163> of user <ken>, submitted from host <apple>
Mon Mar 1 15:00:56 EST 1996:
Job <199> of user <tim>, submitted from host <pear>
```

The following program displays the job arrays of all users in the LSF Batch system and displays the breakdown of jobs as far as job status is concerned. The program demonstrates the use of LSBLIB API calls for collecting summary information of a job array.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

int
main(int argc, char **argv)
{
    struct jobInfoEnt  *job;
    int numJobs;
    int more;

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

numJobs = lsb_openjobinfo(0, NULL, "all", NULL, NULL, ALL_JOB|JGRP_ARRAY_INFO);
if (numJobs < 0) {
    lsb_perror("lsb_openjobinfo");
    exit(-1);
}

printf("JOBID  ARRAY_NAME  OWNER  NJOBS PEND DONE RUN EXIT SSUSP USUSP PSUSP\n");
more = 1;
for (;;) {
```

```
    if (!more)
        break;

    job = lsb_readjobinfo(&more);

    printf("%-5d   %-8.8s ", LSB_ARRAY_JOBID(job->jobId), job->submit.jobName);
    printf("%8.8s ", job->user);

    printf("  %5d %4d %4d %4d %4d %5d %5d %5d\n",
            job->counter[JGRP_COUNT_NJOBS],
            job->counter[JGRP_COUNT_PEND],
            job->counter[JGRP_COUNT_NDONE],
            job->counter[JGRP_COUNT_NRUN],
            job->counter[JGRP_COUNT_NEXIT],
            job->counter[JGRP_COUNT_NSSUSP],
            job->counter[JGRP_COUNT_NUSUSP],
            job->counter[JGRP_COUNT_NPSUSP]);
    }
    lsb_closejobinfo();

    exit(0);
}
```

The above program produces output similar to the following:

```
JOBID    ARRAY_NAME   OWNER     NJOBS PEND DONE  RUN EXIT SSUSP USUSP PSUSP
4205     ja1[1-8]     userA         8    0    0    0    0     0     0     8
4207     ja2[1-2]     userB         2    0    0    0    0     0     0     2
5074     ja3[1-4]     userA         4    0    3    1    0     0     0     0
5075     ja4[1-10]    userC        17    0   13    0    0     4     0     0
5076     ja5[1-4]     userD         4    0    1    0    3     0     0     0
```

Job Manipulation

After a job has been submitted, it can be manipulated by users in different ways. It can be suspended, resumed, killed, or sent an arbitrary signal.

Note

All applications that manipulate jobs are subject to authentication provisions described in 'Authentication' on page 17.

Sending a Signal To a Job

Users can send signals to submitted jobs. If the job has not been started, you can send `KILL`, `TERM`, `INT`, and `STOP` signals. These will cause the job to be cancelled (`KILL`, `TERM`, `INT`) or suspended (`STOP`). If the job is already started, then any signals can be sent to the job.

The LSBLIB call to send a signal to a job is:

`int lsb_signaljob(`*`jobId`*`, `*`sigValue`*`);`

The `jobId` and `sigValue` parameters are self-explanatory.

The following example takes a job ID as the argument and send a `SIGSTOP` signal to the job.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

main(argc, argv)
    int  argc;
    char *argv[];
{
    if (argc != 2) {
        printf(“Usage: %s jobId\n”, argv[0]);
        exit(-1);
    }

    if (lsb_init(argv[0]) < 0) {
        lsb_perror(“lsb_init”);
        exit(-1);
    }

    if (lsb_signaljob(atoi(argv[1]), SIGSTOP) <0) {
        lsb_perror(“lsb_signaljob”);
        exit(-1);
    }

    printf(“Job %d is signaled\n”, argv[1]);
    exit(0);
}
```

Switching a Job To a Different Queue

A job can be switched to a different queue after submission. This can be done even after the job has already started.

The LSBLIB function to switch a job from one queue to another is:

int lsb_switchjob(*jobId*, *queue*);

Below is an example program that switches a specified job to a new queue.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    if (argc != 3) {
        printf("Usage: %s jobId new_queue\n", argv[0]);
        exit(-1);
    }

    if (lsb_init(argv[0]) <0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    if (lsb_switchjob(argv[1], argv[2]) < 0) {
        lsb_perror("lsb_switchjob");
        exit(-1);
    }

    printf("Job %d is switched to new queue <%s>\n", argv[1], argv[2]);

    exit(0);
}
```

Forcing a Job to Run

After a job is submitted to the LSF Batch system, it remains pending until LSF Batch determines that it is ready to run (for details on the factors that govern when and where a job starts to run, see "How LSF Batch Schedules Jobs" in the *LSF Batch Administrator's Guide*). However, a job can be forced to run on a specified list of hosts immediately using the following LSBLIB function:

```
int lsb_runjob(runJobReq)
```

This function takes the `runJobReq` structure which is defined in lsbatch.h:

```
struct runJobReq {
    int jobId;            Job ID of the job to start
    int numHosts;         Number of hosts to run the job on
    char **hostname;      Host names where jobs run
    int options;          RUNJOB_REQ_NORMAL or RUNJOB_REQ_NOSTOP
}
```

A job can be started and run subject to no scheduling constraints, such as job slot limits. If the job is started with the options field being 0 or RUNJOB_REQ_NORMAL, then the job will still be subject to the underlying queue's run windows and to the threshold of the queue and of the job's execution hosts.

To override this, use RUNJOB_REQ_NOSTOP and the job will not be stopped due to the above mentioned load conditions. However, all LSBLIB's job munipulation APIs can still be applied to the job.

The following is an example program that runs a specified job on a host that has no batch job running.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

int
main(int argc, char **argv)
{
    struct hostInfoEnt  *hInfo;
    int numHosts;

    if (argc != 2) {
        printf("Usage: %s jobId\n", argv[0]);
```

```
        exit(-1);
    }

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    hInfo = lsb_hostinfo(NULL, &numHosts);
    if (hInfo == NULL) {
        lsb_perror("lsb_hostinfo");
        exit(-1);
    }

    for (i = 0; i < numHosts; i++) {
        if (hInfo[i].hStatus & (HOST_STAT_BUSY | HOST_STAT_WIND
                                | HOST_STAT_DISABLED | HOST_STAT_LOCKED
                                | HOST_STAT_FULL | HOST_STAT_NOLIM
                                | HOST_STAT_UNLICENSED | HOST_STAT_UNAVAIL
                                | HOST_STAT_UNREACH))
            continue;

        /* found a vacant host */
        if (hInfo[i].numJobs == 0)
            break;
    }

    if (i == numHosts) {
        fprintf(stderr, "Cannot find vacate host to run job < %d >\n",
                jobId);
        exit(-1);
    }

    /* The job can be stopped due to load conditions */
    runJobReq.options = 0;
    runJobReq.numHosts = 1;
    runJobReq.hosts = &hInfo[i].host

    if (lsb_runjob(&runJobReq) < 0) {
        lsb_perror("lsb_runjob");

        exit(-1);
    }
```

```
    exit (0);
}
```

Processing LSF Batch Log Files

LSF Batch saves a lot of valuable information about the system and jobs. Such information is logged by `mbatchd` in files `lsb.events` and `lsb.acct` under the directory `$LSB_SHAREDIR/`*`your_cluster/`*`logdir`, where `LSB_SHAREDIR` is defined in the `lsf.conf` file and *your_cluster* is the name of your LSF cluster.

`mbatchd` logs such information for several purposes. Firstly, some of the events serve as the backup of `mbatchd`'s memory so that in case `mbatchd` crashes, all the critical information can be picked up by the newly started `mbatchd` from the event file to restore the current state of LSF Batch. Secondly, the events can be used to produce historical information about the LSF Batch system and user jobs. Lastly, such information can be used to produce accounting or statistic reports.

CAUTION!

> **The `lsb.events` file contains critical user job information. It should never be modified by your program. Writing into this file may cause the loss of user jobs.**

LSBLIB provides a function to read information from these files into a well-defined data structure:

`struct eventRec *lsb_geteventrec(`*`log_fp`*`, `*`lineNum`*`)`

The parameters are:

`FILE *log_fp;`	File handle for either an event log file or job log file
`nt *lineNum;`	Line number of the next event record

The parameter `log_fp` is as returned by a successful `fopen()` call. The content in `lineNum` is modified to indicate the line number of the next event record in the log file on a successful return. This value can then be used to report the line number when an error occurs while reading the log file. This value should be initiated to 0 before `lsb_geteventrec()` is called for the first time.

This call returns the following data structure:

```
struct eventRec {
    char  version[MAX_VERSION_LEN];    Version number of the mbatchd
    int   type;                        Type of the event
    int   eventTime;                   Event time stamp
    union eventLog eventLog;           Event data
};
```

The event type is used to determine the structure of the data in `eventLog`. LSBLIB remembers the storage allocated for the previously returned data structure and automatically frees it before returning the next event record.

`lsb_geteventrec()` returns `NULL` and sets `lsberrno` to `LSBE_EOF` when there are no more records in the event file.

Events are logged by `mbatchd` for many different purposes. There are job-related events and system-related events. Applications can choose to process certain events and ignore other events. For example, the `bhist` command processes job-related events only. The currently available event types are listed below.

Table 5. Event Types

Event Type	Description
`EVENT_JOB_NEW`	New job event
`EVENT_JOB_START`	`mbatchd` is trying to start a job
`EVENT_JOB_STATUS`	Job status change event
`EVENT_JOB_SWITCH`	Job switched to a new queue
`EVENT_JOB_MOVE`	Job moved within a queue
`EVENT_QUEUE_CTRL`	Queue status changed by LSF admin
`EVENT_HOST_CTRL`	Host status changed by LSF admin
`EVENT_MBD_START`	New mbatchd start event
`EVENT_MBD_DIE`	mbatchd resign event
`EVENT_MBD_UNFULFILL`	mbatchd has an action to be fulfilled
`EVENT_JOB_FINISH`	Job has finished (logged in lsb.acct only)

Table 5. Event Types

Event Type	Description
EVENT_LOAD_INDEX	Complete list of load index names
EVENT_MIG	Job has migrated
EVENT_PRE_EXEC_START	The pre-execution command started
EVENT_JOB_ROUTE	The job has been routed to NQS
EVENT_JOB_MODIFY	The job has been modified
EVENT_JOB_SIGNAL	Job signal to be delivered
EVENT_CAL_NEW	New calendar event [1]
EVENT_CAL_MODIFY	Calendar modified 1
EVENT_CAL_DELETE	Calendar deleted 1
EVENT_JOB_FORCE	Forcing a job to start on specified hosts
EVENT_JOB_FORWARD	Job forwarded to another cluster
EVENT_JOB_ACCEPT	Job from a remote cluster dispatched
EVENT_STATUS_ACK	Job status successfully sent to submission cluster
EVENT_JOB_EXECUTE	Job started successfully
EVENT_JOB_REQUEUE	Job is requeued
EVENT_JOB_SIGACT	An signal action on a job has been initiated or finished
EVENT_JOB_START_ACCEPT	Job accepted by sbatchd

1. Available only if the LSF JobScheduler component is enabled.

Note that the event type `EVENT_JOB_FINISH` is used by the `lsb.acct` file only and all other event types are used by the `lsb.events` file only. For detailed formats of these log files, see `lsb.events(5)` and `lsb.acct(5)`.

Each event type corresponds to a different data structure in the union:

```
union  eventLog {
    struct jobNewLog      jobNewLog;              EVENT_JOB_NEW
    struct jobStartLog    jobStartLog;            EVENT_JOB_START
    struct jobStatusLog   jobStatusLog;           EVENT_JOB_STATUS
    struct jobSwitchLog   jobSwitchLog;           EVENT_JOB_SWITCH
    struct jobMoveLog     jobMoveLog;             EVENT_JOB_MOVE
    struct queueCtrlLog   queueCtrlLog;           EVENT_QUEUE_CTRL
    struct hostCtrlLog    hostCtrlLog;            EVENT_HOST_CTRL
    struct mbdStartLog    mbdStartLog;            EVENT_MBD_START
    struct mbdDieLog      mbdDieLog;              EVENT_MBD_DIE
    struct unfulfillLog   unfulfillLog;           EVENT_MBD_UNFULFILL
    struct jobFinishLog   jobFinishLog;           EVENT_JOB_FINISH
    struct loadIndexLog   loadIndexLog;           EVENT_LOAD_INDEX
    struct migLog         migLog;                 EVENT_MIG
    struct calendarLog    calendarLog;            Shared by all calendar events
    struct jobForce       jobForceRequestLog      EVENT_JOB_FORCE
    struct jobForwardLog jobForwardLog;           EVENT_JOB_FORWARD
    struct jobAcceptLog   jobAcceptLog;           EVENT_JOB_ACCEPT
    struct statusAckLog   statusAckLog;           EVENT_STATUS_ACK
    struct signalLog      signalLog;              EVENT_JOB_SIGNAL
    struct jobExecuteLog jobExecuteLog;           EVENT_JOB_EXECUTE
    struct jobRequeueLog jobRequeueLog;           EVENT_JOB_REQUEUE
    struct sigactLog sigactLog;                   EVENT_JOB_SIGACT
    struct jobStartAcceptLog jobStartAcceptLog    EVENT_JOB_START_ACCEPT
};
```

The detailed data structures in the above union are defined in `lsbatch.h` and described in `lsb_geteventrec(3)`.

Below is an example program that takes an argument as job name and displays a chronological history about all jobs matching the job name. This program assumes that the `lsb.events` file is in `/local/lsf/work/cluster1/logdir`.

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <lsf/lsbatch.h>

main(argc, argv)
    int  argc;
```

```
    char *argv[];
{
    char *eventFile = "/local/lsf/work/cluster1/logdir/lsb.events";
    FILE *fp;
    struct eventRec *recrod;
    int  lineNum = 0;
    char *jobName = argv[1];
    int  i;

    if (argc != 2) {
        printf("Usage: %s jobname\n", argv[0]);
        exit(-1);
    }

    if (lsb_init(argv[0]) < 0) {
        lsb_perror("lsb_init");
        exit(-1);
    }

    fp = fopen(eventFile, "r");
    if (fp == NULL) {
        perror(eventFile);
        exit(-1);
    }

    for (;;) {

        record = lsb_geteventrec(fp, &lineNum);
        if (record == NULL) {
            if (lsberrno == LSBE_EOF)
                exit(0);
            lsb_perror("lsb_geteventrec");
            exit(-1);
        }

        if (strcmp(record->eventLog.jobNewLog.jobName, jobName) != 0)
            continue;

        switch (record->type) {
            struct jobNewLog *newJob;
            struct jobStartLog *startJob;
            struct jobStatusLog *statusLog;

        case EVENT_JOB_NEW:
```

```
      newJob = &(record->eventLog.jobNewLog);
    printf("%s: job <%d> submitted by <%s> from <%s> to <%s> queue\n",
        ctime(&record->eventTime), newJob->jobId, newJob->userName,
        newJob->fromHost, newJob->queue);
    continue;
case EVENT_JOB_START:
    startJob = &(record->eventLog.jobStartLog);
    printf("%s: job <%d> started on ",
                ctime(&record->eventTime), newJob->jobId);
    for (i=0; i<startJob->numExHosts; i++)
        printf("<%s> ", startJob->execHosts[i]);
    printf("\n");
    continue;
case EVENT_JOB_STATUS:
    statusJob = &(record->eventLog.jobStatusLog);
    printf("%s: Job <%d> status changed to: ",
                ctime(&record->eventTime), statusJob->jobId);
    switch(statusJob->jStatus) {
    case JOB_STAT_PEND:
        printf("pending\n");
        continue;
    case JOB_STAT_RUN:
        printf("running\n");
        continue;
    case    JOB_STAT_SSUSP:
    case JOB_STAT_USUSP:
    case JOB_STAT_PSUSP:
        printf("suspended\n");
        continue;
    case JOB_STAT_UNKWN:
        printf("unknown (sbatchd unreachable)\n");
        continue;
    case JOB_STAT_EXIT:
        printf("exited\n");
        continue;
    case JOB_STAT_DONE:
        printf("done\n");
        continue;

    default:
        printf("\nError: unknown job status %d\n", statusJob->jStatus);
        continue;
    }
default:            /* Only display a few selected event types*/
```

```
            continue;
        }
    }

    exit(0);
}
```

Note that in the above program, events that are of no interest are skipped. The job status codes are defined in `lsbatch.h`. The `lsb.acct` file stores job accounting information and can be processed similarly. Since currently there is only one event type (`EVENT_JOB_FINISH`) in `lsb.acct` file, the processing is simpler than the above example.

4 Advanced Programming Topics

LSF API provides flexibility for programmers to write complex load sharing applications. Previous chapters covered the basic programming techniques using LSF APIs. This chapter will look into a few more advanced topics in LSF application programming.

Both LSLIB and LSBLIB are used in the examples of this chapter.

Getting Load Information on Selected Load Indices

‘Getting Dynamic Load Information’ on page 28 showed an example that gets load information from the LIM. Depending on the size of your LSF cluster and the frequency at which the `ls_load()` function is called, returning the load information about all hosts can produce unnecessary overhead to hosts and network.

LSLIB provides a function call that will allow an application to specify a selective number of load indices and get only those load indices that are of interest to the application.

Getting a List of All Load Index Names

Since LSF allows a site to install an ELIM (External LIM) to collect additional load indices, the names and the total number of load indices are often dynamic and have to be found out at run time unless the application is only using the built-in load indices.

Below is an example routine that returns a list of all available load index names and the total number of load indices.

```
#include <lsf/lsf.h>

char **getIndexList(listsize)
    int *listsize;
{
    struct lsInfo *lsInfo;
    static char *nameList[MAXLOADINDEX];
    static int first = 1;

    if (first) {       /* only need to do so when called for the first time */
        lsInfo = ls_info();
        if (lsInfo == NULL)
            return (NULL);
        first = 0;
    }

    if (listSize != NULL)
        *listSize = lsInfo->numIndx;

    for (i=0; i<lsInfo->numIndx; i++)
        nameList[i] = lsInfo->resTable[i].name;

    return (nameList);
}
```

The above routine returns a list of load index names currently installed in the LSF cluster. The content of `listSize` will be modified to the total number of load indices. The program would return `NULL` if the `ls_info()` function fails. The data structure returned by `ls_info()` contains all the load index names before any other resource names. The load index names start with the 11 built-in load indices followed by site external load indices (through ELIM).

Displaying Selected Load Indices

By providing a list of load index names to an LSLIB function, you can get the load information about the specified load indices.

The following example shows how you can display the values of the external load indices. This program uses the following LSLIB function:

```
struct hostLoad *ls_loadinfo(resreq, numhosts, options, fromhost,
                              hostlist, listsize, namelist)
```

The parameters for this routine are:

`char *resreq;`	Resource requirement
`int *numhosts;`	Return parameter, number of hosts returned
`int options;`	Host and load selection options
`char *fromhost;`	Used only if DFT_FROMTYPE is set in options
`char **hostlist;`	A list of candidate hosts for selection
`int listsize;`	Number of hosts in hostlist
`char ***namelist;`	Input/output parameter -- load index name list

This call is similar to `ls_load()` except that it allows an application to supply both a list of load indices and a list of candidate hosts. If both these parameters are `NULL`, then it is exactly the same as `ls_load()` function.

The parameter `namelist` allows an application to specify a list of load indices of interest. the function then returns only the specified load indices. On return this parameter is modified to point to another name list that contains the same set of load index names, but in a different order to reflect the mapping of index names and the actual load values returned in the `hostLoad` array:

```
#include <stdio.h>
#include <lsf.lsf.h>

main()
{
    struct hostLoad *load;
    char **loadNames;
    int numIndx;
    int numUsrIndx;
    int nHosts;

    loadNames = getIndexList(&numIndx);
    if (loadNames == NULL) {
        ls_perror("Unable to get load index names\n");
        exit(-1);
    }
```

```
    numUsrIndx = numIndx - 11;  /* this is the total num of site defined indices*/
    if (numUsrIndx == 0) {
        printf("No external load indices defined\n");
        exit(-1);
    }

    loadNames += 11;              /* skip the 11 built-in load index names */

    load = ls_loadinfo(NULL, &nHosts, 0, NULL, NULL, 0, &loadNames);
    if (load == NULL) {
        ls_perror("ls_loadinfo");
        exit(-1);
    }

    printf("Report on external load indices\n");

    for (i=0; i<nHosts; i++) {
        printf("Host %s:\n", load[i].hostName);
        for (j=0; j<numUsrindx; j++)
            printf("            index name: %s, value %5.0f\n",
                   loadNames[j], load[i].li[j]);
    }
}
```

The above program uses the `getIndexList()` function described in the previous example program to get a list of all available load index names. Sample output from the above program follows:

```
Report on external load indices
Host hostA:
        index name: usr_tmp, value 87
        index name: num_licenses, value 1
Host hostD:
        index name: usr_tmp, value 18
        index name: num_licenses, value 2
```

Writing a Parallel Application

LSF provides job placement and remote execution support for parallel applications. LIM's host selection or placement service can return an array of good hosts for an application. The application can then use remote execution service provided by RES to run tasks on these hosts concurrently.

In this section are examples of writing a parallel application using LSLIB.

`ls_rtask()` Function

‘Running a Task Remotely’ on page 43 discussed the use of `ls_rexecv()` function for remote execution. There is another LSLIB call for remote execution: `ls_rtask()`. These two functions differ in how the client side behaves.

The `ls_rexecv()` is useful when local side does not need to do anything but wait for the remote task to finish. After initiating the remote task, `ls_rexecv()` replaces the current program with the Network I/O Server (NIOS) by calling the `execv()` system call. The NIOS then handles the rest of the work on the local side: delivering input/output between local terminal and remote task and exits with the same status as the remote task. `ls_rexecv()` may be considered as the remote execution version of the UNIX `execv()` system call.

`ls_rtask()` provides more flexibility if the client side has to do other things after the remote task is initiated. For example, the application may want to start more than one task on several hosts. Unlike `ls_rexecv()`, `ls_rtask()` returns immediately after the remote task is started. The syntax of `ls_rtask()` is:

```
int ls_rtask(host, argv, options)
```

The parameters are:

`char *host;`	Name of the remote host to start task on
`char **argv;`	Program name and arguments
`int options;`	Remote execution options

The `options` parameter is similar to that of the `ls_rexecv()` function. This function returns the task ID of the remote task which can then be used by the application to differentiate possibly multiple outstanding remote tasks. When a remote task finishes, the status of the remote task is sent back to the NIOS running on the local host, which then notifies the application by issuing a `SIGUSR1` signal. The application can then call `ls_rwait()` to collect the status of the remote task. The `ls_rwait()` behaves in much the same way as the `wait(2)` system call. `ls_rtask()` may be considered as a combination of remote `fork()` and `execv()`.

Note

Applications calling `ls_rtask()` *must set up signal handler for the* `SIGUSR1` *signal, or the application could be killed by* `SIGUSR1`.

You need to be careful if your application handles `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal. If handlers for these signals are `SIG_DFL`, the `ls_rtask()` function automatically installs a handler for them to properly coordinate with the NIOS when these signals are received. If you intend to handle these signals by yourself instead of using the default set by LSLIB, you need to use the low level LSLIB function `ls_stoprex()` before the end of your signal handler.

Running Tasks on Many Machines

Below is an example program that uses `ls_rtask()` to run `rm -f /tmp/core` on user specified hosts.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <lsf/lsf.h>

main (argc, argv)
    int argc;
    char *argv[];
{
    char *command[4];
    int numHosts;
    int i;
    int tid;

    if (argc <= 1) {
        printf(“Usage: %s host1 [host2 ...]\n”);
        exit(-1);
    }

    numHosts = argc - 1;
    command[0] = “rm”;
    command[1] = “-f”;
    command[2] = “/tmp/core”;
```

```
    command[3] = NULL;

    if (ls_initrex(numHosts, 0) < 0) {
        ls_perror("ls_initrex");
        exit(-1);
    }

    signal(SIGUSR1, SIG_IGN);

    /* Run command on the specified hosts */
    for (i=1; i<=numHosts; i++) {
        if ((tid = ls_rtask(argv[i], command, 0)) < 0) {
            fprintf(stderr, "lsrtask failed for host %s: %s\n",
                    argv[i], ls_sysmsg());
            exit(-1);
        }
        printf("Task %d started on %s\n", tid, argv[i]);
    }

    while (numHosts) {
        LS_WAIT_T status;

        tid = ls_rwait(&status, 0, NULL);
        if (tid < 0) {
            ls_perror("ls_rwait");
            exit(-1);
        }

        printf("task %d finished\n", tid);
        numHosts--;
    }

    exit(0);
}
```

The above program set the signal handler for `SIGUSR1` to `SIG_IGN`. This causes the signal to be ignored. It uses `ls_rwait()` to poll the status of remote tasks. You could set a signal handler so that it calls `ls_rwait()` inside the signal handler.

The task ID could be used to preform an operation on the task. For example, you can send a signal to a remote task explicitly by calling `ls_rkill()`.

If you want to run the task on remote hosts one after another, instead of concurrently, you can call `ls_rwait()` right after `ls_rtask()`.

Also note the use of `ls_sysmsg()` instead of `ls_perror()`, which does not allow flexible printing format.

The above example program produces output similar to the following:

```
% a.out hostD hostA hostB
Task 1 started on hostD
Task 2 started on hostA
Task 3 started on hostB
Task 1 finished
Task 3 finished
Task 2 finished
```

Note that remote tasks are run concurrently, so the order in which tasks finish is not necessarily the same as the order in which tasks are started.

Finding out Why the Job Is Still Pending

‘Getting Information about Batch Jobs’ on page 63 showed how to get information about submitted jobs. It is frequently desirable to know the reasons why jobs are in certain status. The LSBLIB provides a function to print such information. This section describes a routine that prints out why a job is in pending status.

When `lsb_readjobinfo()` reads a record of a pending job, the variables `reasons` and `subreasons` contained in the returned `jobInfoEnt` data structure can be used to call the following LSBLIB function to get the reason text explaining why the job is still in pending state:

char *lsb_pendreason(*pendReasons*, *subReasons*, *ld*)

where `pendReasons` and `subReasons` are integer reason flags as returned by a `lsb_readjobinfo()` function while `ld` is a pointer to the following data structure:

```
struct loadIndexLog {
    int nIdx;                    Number of load indices configured for the LSF cluster
```

```
    char **name;                    List of the load index names
}
```

The example program below should be called by your application after `lsb_readjobinfo()` is called.

```
#include <stdio.h>
#include <lsf/lsbatch.h>

char *
reasonToText(reasons, subreasons)
    int reasons;
    int subreasons;
{
    struct loadIndexLog indices;

    /* first get the list of all load index names */
    indices.name = getIndexList(&indices.nIdx);

    return (lsb_pendreason(reasons, subreasons, &indices));

}
```

A similar routine can be written to print out the reason why a job was suspended. The corresponding LSBLIB call is:

`char *lsb_suspreason(`*reasons*`, `*subreasons*`, `*ld*`)`

The parameters for this function are the same as those for the `lsb_pendreason()` function.

Reading `lsf.conf` Parameters

It is frequently desirable for your applications to read the contents of the `lsf.conf` file or even define your own site specific variables in the `lsf.conf` file.

The `lsf.conf` file follows the syntax of Bourne shell, and therefore could be sourced by a shell script and set into your environment before starting your C program. Your program can then get these variables as environment variables.

LSLIB provides a function to read the `lsf.conf` variables in your C program:

```
int ls_readconfenv(paramList, confPath)
```

where `confPath` is the directory in which the `lsf.conf` file is stored. `paramList` is an array of the following data structure:

```
struct config_param {
    char *paramName;                        Name of the parameter, input
    char *paramValue;                       Value of the parameter, output
}
```

`ls_readconfenv()` reads the values of the parameters defined in `lsf.conf` matching the names described in the `paramList` array. Each resulting value is saved into the `paramValue` variable of the array element matching `paramName`. If a particular parameter mentioned in the paramList is not defined in `lsf.conf`, then on return its value is left `NULL`.

The following example program reads the variables `LSF_CONFDIR`, `MY_PARAM1`, and `MY_PARAM2` in `lsf.conf` file and displays them on screen. Note that `LSF_CONFDIR` is a standard LSF parameter, while the other two parameters are user site-specific. It assumes `lsf.conf` is in `/etc` directory.

```
#include <stdio.h>
#include <lsf/lsf.h>

struct config_param myParams[] =
{
#define LSF_CONFDIR                     0
     {“LSF_CONFDIR”, NULL},
#define MY_PARAM1                       1
     {“MY_PARAM1”, NULL),
#define MY_PARAM2                       2
     {“MY_PARAM2”, NULL),
     {NULL, NULL}
}

main()
{
    if (ls_readconfenv(myParams, “/etc”) < 0) {
        ls_perror(“ls_readconfenv”);
        exit(-1);
```

```
    }

    if (myParams[LSF_CONFDIR].paramValue == NULL)
        printf("LSF_CONFDIR is not defined in /etc/lsf.conf\n");
    else
        printf("LSF_CONFDIR=%s\n", myParams[LSF_CONFDIR].paramValue);

    if (myParams[MY_PARAM1].paramValue == NULL)
        printf("MY_PARAM1 is not defined in /etc/lsf.conf\n");
    else
        printf("MY_PARAM1=%s\n", myParams[MY_PARAM1].paramValue);

    if (myParams[MY_PARAM2].paramValue == NULL)
        printf("MY_PARAM2 is not defined\n");
    else
        printf("MY_PARAM2=%s\n", myParams[MY_PARAM2].paramValue);

    exit(0);
}
```

The `paramValue` parameter in the `config_param` data structure must be initialized to `NULL` and is then modified to point to a result string if a matching `paramName` is found in the `lsf.conf` file. The array must end with a `NULL` `paramName`.

Signal Handling in Windows NT

LSF uses the UNIX signal mechanism to perform job control. For example, the bkill command in UNIX normally results in the signals SIGINT, SIGTERM, and SIGKILL being sent to the target job. Signal-handling code that already exists in the in UNIX applications allows them to shut down gracefully, in stages. In the past, the same bkill command in Windows NT has been accomplished by a call to TerminateProcess(), which terminates the application immediately and does not allow it to release shared resources or clean up the way a UNIX application can.

LSF version 3.2 has been modified to provide signal notification through the Windows NT message queue. LSF now includes messages corresponding to common UNIX signals. This means that a customized Windows NT application can process these messages.

For example, the bkill command now sends the SIGINT and SIGTERM signals to Windows NT applications as job control messages. An LSF-aware Windows NT application can interpret these messages and shut down neatly.

To write a Windows NT application that takes advantage of this feature, register the specific signal messages that the application will handle. Then modify the message loop to check each message before dispatching it, and take the appropriate action if it is a job control message.

The following examples show sample code that might help you to write your own applications.

Job Control in a Windowed Application

This is an example program showing how a windowed application can receive NT job control notification from the LSF system.

Catching the notification messages involves:

1) Registering the windows messages for the signal(s) that you want to receive (in this case, SIGTERM).
2) In your GetMessage loop, looking for the message(s) you want to catch.

Note that you can't DispatchMessage() the message, since it is addressed to the thread, not the window. This program just displays some information in its main window, and waits for SIGTERM. Once SIGTERM is received, it posts a quit message and exits. A real program could do some cleanup when the SIGTERM message is received.

```
/* WINJCNTL.C */

#include <windows.h>
#include <stdio.h>

#define BUFSIZE 512

static UINT msgSigTerm;

static int xpos;
static int pid_ypos;
static int tid_ypos;
static int msg_ypos;
```

```
static int pid_buf_len;
static int tid_buf_len;
static int msg_buf_len;
static char pid_buf[BUFSIZE];
static char tid_buf[BUFSIZE];
static char msg_buf[BUFSIZE];

LRESULT WINAPI MainWndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{

    HDC hDC;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;
    switch (msg) {

        case WM_CREATE:

            hDC = GetDC(hWnd);
            GetTextMetrics(hDC, &tm);
            ReleaseDC(hWnd, hDC);
            xpos = 0;
            pid_ypos = 0;
            tid_ypos = pid_ypos + tm.tmHeight;
            msg_ypos = tid_ypos + tm.tmHeight;
            break;

        case WM_PAINT:

            hDC = BeginPaint(hWnd, &ps);
            TextOut(hDC, xpos, pid_ypos, pid_buf, pid_buf_len);
            TextOut(hDC, xpos, tid_ypos, tid_buf, tid_buf_len);
            TextOut(hDC, xpos, msg_ypos, msg_buf, msg_buf_len);
            EndPaint(hWnd, &ps);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;


        default:
            return DefWindowProc(hWnd, msg, wParam, lParam);
    }
```

```
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)

{
    ATOM rc;
    WNDCLASS wc;
    HWND hWnd;
    MSG msg;

/* Create and register a windows class */

    if (hPrevInstance == NULL) {
            wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
            wc.lpfnWndProc = MainWndProc;
            wc.cbClsExtra = 0;
            wc.cbWndExtra = 0;
            wc.hInstance = hInstance;
            wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
            wc.hCursor = LoadCursor(NULL, IDC_ARROW);
            wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);

            rc = RegisterClass(&wc);
    }

/* Register the message we want to catch */

    msgSigTerm = RegisterWindowMessage("SIGTERM");

/* Format some output for the main window */
sprintf(pid_buf, "My process ID is: %d", GetCurrentProcessId());
pid_buf_len = strlen(pid_buf);
sprintf(tid_buf, "My thread ID is: %d", GetCurrentThreadId());
tid_buf_len = strlen(tid_buf);
sprintf(msg_buf, "Message ID is: %u", msgSigTerm);
msg_buf_len = strlen(msg_buf);

/* Create the main window */

    hWnd = CreateWindow("WinJCntlClass",
            "Windows Job Control Demo App",
            WS_OVERLAPPEDWINDOW,
```

```
            0,
            0,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            NULL,
            NULL,
            hInstance,
            NULL);
    ShowWindow(hWnd, nCmdShow);

/* Enter the message loop, waiting for msgSigTerm. When we get it, just post a
quit message */

    while (GetMessage(&msg, NULL, 0, 0)) {
        if (msg.message == msgSigTerm) {
            PostQuitMessage(0);
        } else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}
```

Job Control in a Console Application

This is an example program showing how a console application can receive NT job control notification from the LSF system.

Catching the notification messages involves:

1) Registering the windows messages for the signals that you want to receive (in this case, SIGINT and SIGTERM).
2) Creating a message queue by calling PeekMessage (this is how Microsoft suggests console apps should create message queues).
3) Enter a GetMessage loop, looking for the message you want to catch.

Note that you don’t DispatchMessage here, since you don’t have a window to dispatch to.

This program just sits in the message loop, waiting for SIGINT and SIGTERM, and displays messages when those signals are received. A real application would do clean-up and exit if it received either of these signals.

```
/* CONJCNTL.C */

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)

{
    DWORD pid = GetCurrentProcessId();
    DWORD tid = GetCurrentThreadId();
    UINT msgSigInt = RegisterWindowMessage("SIGINT");
    UINT msgSigTerm = RegisterWindowMessage("SIGTERM");
    MSG msg;

/* Make a message queue -- this is the method suggested by MS */

    PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
    printf("My process id: %d\n", pid);
    printf("My thread id: %d\n", tid);
    printf("SIGINT message id: %d\n", msgSigInt);
    printf("SIGTERM message id: %d\n", msgSigTerm);
    printf("Entering loop...\n");
    fflush(stdout);

    while (GetMessage(&msg, NULL, 0, 0)) {
        printf("Received message: %d\n", msg.message);
        if (msg.message == msgSigInt) {
            printf("SIGINT received, continuing.\n");
        } else if (msg.message == msgSigTerm) {
            printf("SIGTERM received, continuing.\n");
        }
        fflush(stdout);
    }

    printf("Exiting.\n");
    fflush(stdout);
    return EXIT_SUCCESS;
}
```

A List of LSF API Functions

This appendix lists all the LSF API functions for your reference. Many of the functions listed below are not documented in this guide, but are described in detail in the on-line man pages. See `lslib(3)` and `lsblib(3)` for details of these functions.

LSLIB Functions

These are the function calls provided by the LSF base system API. The function calls are listed by service categories.

Cluster Configuration Information

`struct lsInfo *ls_info(void)`
Get cluster-wide configuration information.

`char *ls_getclustername(void)`
Get the name of the local cluster.

`char *ls_getmastername(void)`
Get the name of the master host.

`float *ls_getmodelfactor(char *modelname)`
Get the CPU factor of the given host model.

`char *ls_gethosttype(char *hostname)`
Get the host type of the given host.

`char *ls_gethostmodel(char *hostname)`
Get the host model of the given host.

```
float *ls_gethostfactor(char *hostname)
```
Get the CPU factor of the given host.

```
struct hostInfo *ls_gethostinfo(char *resreq, int *numhosts,
      char **hostlist, int listsize, int options)
```
Get host related configuration information.

```
int ls_readconfenv(struct config_param *paramList,
      char *confPath)
```
Get the variables defined in `lsf.conf`.

Load Information and Placement Advice

```
struct hostLoad *ls_load(char *resreq, int *numhosts,
      int options, char *fromhost)
```
Get load information of qualified hosts, simple version.

```
struct hostLoad *ls_loadinfo(char *resreq, int *numhosts,
      int options, char *fromhost, char **hostlist,
      int listsize, char ***indxnamelist)
```
Get load information of qualified hosts, generic version.

```
struct hostLoad *ls_loadofhosts(char *resreq, int *numhosts,
      int options, char *fromhost, char **hostlist,
      int listsize)
```
Get load information of the qualified hosts from the given list of hosts.

```
struct hostLoad *ls_loadoftype(char *resreq, int *numhosts,
      int options, char *fromhost, char *hosttype)
```
Get load information about hosts of the given host type.

```
char **ls_placereq(char *resreq, int *numhosts, int options,
      char *fromhost)
```
Get the best qualified hosts.

```
char **ls_placeofhosts(char *resreq, int *numhosts,
      int options, char *fromhost, char **hostlist,
      int listsize)
```
Get the best qualified hosts from the given list of hosts.

```
char **ls_placeoftype(char *resreq, int *numhosts, int options,
      char *fromhost, char *hosttype)
```

Get the best qualified hosts with the given host type.

```
int ls_loadadj(char *resreq, struct placeInfo *hostlist,
      int listsize)
```

Adjust the load of the given host(s).

Task List Manipulation

```
char *ls_resreq(char *task)
```

Get resource requirements of task in the remote task list.

```
int ls_eligible(char *task, char *resreqstr, char mode)
```

Get resource requirements of `task` in the task list indicated by `mode`.

```
int ls_insertrtask(char *task)
```

Insert `task` into the remote `task` list.

```
int ls_insertltask(char *task)
```

Insert `task` into the local task list.

```
int ls_deletertask(char *task)
```

Remove `task` from the remote task list.

```
int ls_deleteltask(char *task)
```

Remove `task` from the local task list.

```
int ls_listrtask(char ***taskList, int sortflag)
```

Get all tasks in the remote task list.

```
int ls_listltask (char ***taskList, int sortflag)
```

Get all tasks in the local task list.

Remote Execution and Task Control

These functions are subject to the authentication protocols described in ‘Authentication’ on page 17.

```
int ls_initrex(int numPorts, int options)
```
Initialize for remote execution or file operation.

```
int ls_connect(char *hostname)
```
Establish a connection with a remote RES.

```
int ls_rexecv(char *host, char **argv, int options)
```
Remote `execv(2)`. Execute `argv` on `host` with the local environment.

```
int ls_rexecve(char *host, char **argv, int options,
      char **envp)
```
Remote `execve(2)`. Execute `argv` on `host` with the given environment.

```
int ls_rtask(char *host, char **argv, int options)
```
Start `argv` on host with local environment.

```
int ls_rtaske(char *host, char **argv, int options,
      char **envp)
```
Start `argv` on `host` with the given environment.

```
int ls_rwait(LS_WAIT_T *status, int options, struct rusage *ru)
```
Remote `wait(2)`.

```
int ls_rwaittid(int tid, LS_WAIT_T *status, int options,
      struct rusage *ru)
```
Remote `waitpid`(2).

```
int ls_rkill(int tid, int sig)
```
Remote `kill(2)`.

```
int ls_rsetenv(char *host, char **envp)
```
Reset the environment for remote tasks on `host`..

```
int ls_chdir(char *host, char *clntdir)
```
Set the working directory for remote tasks on `host`..

```
int ls_stoprex(void)
```
Inform the NIOS to suspend itself and restore local tty settings.

Remote File Operation

These functions are subject to the authentication protocols described in 'Authentication' on page 17.

`int ls_ropen (char *host, char *fn, int flags, int mode)`
Remote `open(2)` on `host`.

`int ls_rclose(int rfd)`
Remote `close(2)` on `host`.

`int ls_rwrite(int rfd, char *buf, int len)`
Remote `write(2)` on `host`.

`int ls_rread(int rfd, char *buf, int len)`
Remote `read(2)` on `host`.

`off_t ls_rlseek(int rfd, off_t offset, int whence)`
Remote `lseek(2)` on `host`.

`int ls_rfstat(int rfd, struct stat *buf)`
Remote `fstat(2)` on `host`.

`int ls_rstat(char *host, char *fn, struct stat *buf)`
Remote `stat(2)` on `host`.

`int ls_getmnthost(char *file)`
Returns the host name of the file server for `file`.

`char *ls_rgetmnthost(char *host, char *file)`
Return the host name of the file server for `file` on `host`.

`int ls_rfcontrol(int command, int arg)`
Control the behavior of remote file operations.

Administration Operation

`int ls_lockhost(time_t duration)`

Set LIM status of the local host to “locked” for `duration` seconds. The application must be a setuid to root program to use this function.

`int ls_unlockhost(void)`

Cancel a previous lock operation. The application must be a setuid to root program to use this function.

`int ls_limcontrol(char *hostname, int opCode)`

Perform a LIM administration operation as specified by `opCode`. The application must be a setuid to root program to use this function.

`int ls_rescontrol(char *host, int opCode, int options)`

Perform a RES administrative operation as specified by `opCode`. The use of this function is subject to authentication protocols described in ‘Authentication’ on page 17.

Error Handling

`void ls_perror(char *usrMsg)`

Print `usrMsg` followed by the LSLIB error message associated with `lserrno`.

`char *ls_sysmsg(void)`

Return the LSLIB error message associated with `lserrno`.

`void ls_errlog(FILE *fp, const char *fmt, ...)`

Logging an LSLIB error message with time stamp.

Miscellaneous

`int ls_fdbusy(int fd)`

Test if a file descriptor `fd` is in use or reserved by LSF.

LSBLIB Functions

These are function calls provided by the LSF Batch system API. The functions are listed by service categories.

Initialization

```
lsb_init(char *appName)
```
Initialize an LSF Batch application.

LSF Batch System Information

```
struct groupInfoEnt *lsb_hostgrpinfo(char **groups,
      int *numGroups, int options)
```
Get membership of the LSF Batch host groups.

```
struct groupInfoEnt *lsb_usergrpinfo(char **groups,
      int *numGroups, int options)
```
Get membership of the LSF Batch user groups.

```
struct parameterInfo *lsb_parameterinfo(char **names,
      int *numUsers, int options)
```
Get the LSF Batch cluster parameters.

```
struct hostInfoEnt *lsb_hostinfo(char **hosts, int *numHosts)
```
Get information about the LSF Batch server hosts or host groups.

```
struct userInfoEnt *lsb_userinfo(char **users, int *numUsers)
```
Get system information about the LSF Batch users and user groups.

```
struct hostPartInfoEnt *lsb_hostpartinfo(char **hostParts,
      int *numHostParts)
```
Get information about the LSF Batch host partitions.

```
.struct queueInfoEnt *lsb_queueinfo(char **queues,
      int *numQueues, char *host, char *userName, int options)
```
Get information about the LSF Batch queues.

Job Manipulation

These functions are subject to the authentication protocols described in ‘Authentication’ on page 17.

```
int lsb_submit(struct submit *jobSubReq,
      struct submitReply *jobSubReply)
```
Submit a job to the LSF Batch system.

```
int lsb_modify(struct submit *jobSubReq,
      struct submitReply *jobSubReply, int jobId)
```
Change the attributes of an already submitted job.

```
int lsb_signaljob(int jobId, int sigValue)
```
Send job `jobId` signal `sigValue`.

```
int lsb_chkpntjob (int jobId, time_t period, int options)
```
Checkpoint the job `jobId`.

```
int lsb_deletejob (int jobId, int times, int options)
```
Delete a calendar-driven job.

```
int lsb_mig(struct submig *mig, int *badHostIdx)
```
Migrate a job from one host to another.

```
int lsb_movejob(int jobId, int *position, int opCode)
```
Change the position of a pending job within its queue.

```
int lsb_switchjob(int jobId, char *queue)
```
Switch a job `jobId` to queue `queue`.

Job Information

```
int lsb_openjobinfo(int jobId, char *jobName, char *user,
      char *queue, char *host, int options)
```
Open a job information stream for the matching job(s) with `mbatchd`.

```
struct jobInfoEnt *lsb_readjobinfo(int *more)
```
Read a job record from the opened job information stream.

```
void lsb_closejobinfo(void)
```
Close a job information stream.

```
char *lsb_suspreason(int reasons, int subreasons,
      struct loadIndexLog *ld)
```
Convert suspending reason codes into text.

```
char *lsb_pendreason(int reasons, int subreasons,
      struct loadIndexLog *ld)
```
Convert pending reason codes into text.

```
char *lsb_peekjob(int jobId)
```
Get the name of the job's buffered output file. This function is subject to the authentication protocols described in 'Authentication' on page 17.

Event File Processing

```
struct eventRec *lsb_geteventrec(FILE *log_fp, int *lineNum)
```
Read an event record from the opened log file.

LSF Batch Administration

These functions are subject to the authentication protocols described in 'Authentication' on page 17.

```
int lsb_reconfig(void)
```
Reconfigure the LSF Batch system using the current configuration files.

```
int lsb_hostcontrol(char *host, int opCode)
```
Open, close `host` for batch jobs, or restart, shut down `sbatchd` on `host`.

```
int lsb_queuecontrol(char *queue, int opCode)
```
Change the status of an LSF Batch queue.

Calendar Manipulation

These functions can be used only if the LSF JobScheduler component is enabled.

```
int lsb_calendarop(int opCode, int numNames, char **names,
      char *desc, char *timeEvents, int options, char **badStr)
```
Add, modify, or delete a calendar.

```
struct calendarInfoEnt *lsb_calendarinfo(char **calendars,
      int *numCalendars, char *user)
```
Get calendar information.

Error Handling

```
void lsb_perror(char *usrMsg)
```
Print the LSBLIB error message associated with `lsberrno` together with `usrMsg`.

```
char *lsb_sysmsg (void)
```
Return the LSBLIB error message associated with `lsberrno`.

Index

G

H

I

J

L